SYSTEMS ENGINEERING
Research Center

Evaluation of Systems Engineering Methods, Processes and Tools on Department of Defense and Intelligence Community Programs - Phase II

Final Technical Report SERC-2009-TR-004

December 15, 2009

Principal Investigator: Richard Turner – Stevens Institute of Technology Co-
Principal Investigator: Forrest Shull – Fraunhofer Center, University of Maryland

Team Members
Barry Boehm - Professor, University of Southern California
Anne Carrigy - Graduate Student, Stevens Institute of Technology
Lori Clarke - Senior Researcher, University of Massachusetts at Amherst
Paul Componation - Senior Researcher, University of Alabama in Huntsville
Cihan Dagli - Senior Researcher, Missouri University of Science and Technology
Jo Ann Lane - Senior Researcher, University of Southern California
Lucas Layman - Senior Researcher, Fraunhofer Center
Ann Miller - Senior Researcher, Missouri University of Science and Technology
Sue O'Brien - Senior Researcher, University of Alabama in Huntsville
Leon Osterweil - Senior Researcher, University of Massachusetts at Amherst
Dawn Sabados - Senior Researcher, University of Alabama in Huntsville
Sandy Wise - Senior Researcher, University of Massachusetts at Amherst

Contract Number: H98230-08-D-0171 DO001, TO002. RT009

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **15 DEC 2009**	2. REPORT TYPE **Final**	3. DATES COVERED
4. TITLE AND SUBTITLE **Evaluation of Systems Engineering Methods, Processes and Tools on Department of Defense and Intelligence Community Programs - Phase II**		5a. CONTRACT NUMBER **H98230-08-D-0171**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Turner /Dr. Richard**		5d. PROJECT NUMBER **RT 09**
		5e. TASK NUMBER **DO 001 TO 002**
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Stevens Institute of Technology University of Southern California University of Massachusetts at Amherst University of Alabama in Huntsville Missouri University of Science and Technology, Fraunhofer Center at University of Maryland**		8. PERFORMING ORGANIZATION REPORT NUMBER **SERC-2009-TR-004**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **DASD (SE)**		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the results from the second in a series of related efforts to address systems engineering shortfalls in projects characterized as quick response, network enabled, or emergent. The objectives of this task were to 1) Gather additional information on methods, processes and tools (MPTs) associated with the environment identified in Phase 1 of this work and develop a taxonomy of MPTs identified; 2) Investigate the use of micro-process modeling techniques to support the definition and evaluation of MPTs; and, 3) Provide implementation guidance on the three MPTs recommended in Phase 1. The products of the research are directly relevant to the challenges currently being faced by the sponsor œ The description of the three recommended MPTs in an expanded taxonomy and individual implementation guidance œ The development of a micro-process model of Scrum in Little-JIL and successful demonstration of fault tree and finite state verification analyses œ The identification of key critical success factors for rapid response and innovative development environments The recommendations for future research based on these results are also provided in this report.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**86**	

This page intentionally left blank

ABSTRACT

This report describes the results from the second in a series of related efforts to address systems engineering shortfalls in projects characterized as quick response, network-enabled, or emergent. The objectives of this task were to 1) Gather additional information on methods, processes and tools (MPTs) associated with the environment identified in Phase 1 of this work and develop a taxonomy of MPTs identified; 2) Investigate the use of micro-process modeling techniques to support the definition and evaluation of MPTs; and, 3) Provide implementation guidance on the three MPTs recommended in Phase 1.

The products of the research are directly relevant to the challenges currently being faced by the sponsor:

- the description of the three recommended MPTs in an expanded taxonomy and individual implementation guidance
- the development of a micro-process model of Scrum in Little-JIL and successful demonstration of fault tree and finite state verification analyses
- the identification of key critical success factors for rapid response and innovative development environments

The recommendations for future research based on these results are:

- continue the identification of useful MPTs and their description in the expanded taxonomy
- investigate the practicality and usefulness of a test bed facility to evaluate incremental improvement of existing MPTs and new approaches to systems engineering, including new MPTs
- conduct empirical studies of anecdotal MPT claims, e.g. scalability of Scrum
- use the gaps identified in this work to establish focused innovation teams to create, evaluate and if appropriate, pilot new systems engineering approaches and MPTs that will address each team's specific gap
- investigate new process improvement methods that are applicable in rapid response and innovative environments
- identify the characteristics of agile, adaptable processes for agile engineering

This page intentionally left blank

TABLE OF CONTENTS

FIGURES AND TABLES

This page intentionally left blank

1 SUMMARY

This research task is the second in a series of related efforts to address systems engineering shortfalls in projects characterized as quick response, network-enabled, or emergent. The objectives of this task were to 1) Gather additional information on methods, processes and tools (MPTs) associated with the environment identified in Phase 1 of this work and develop a taxonomy of MPTs identified; 2) Investigate the use micro-process modeling techniques to support definition and evaluation of MPTs; and, 3) Provide implementation guidance on the three MPTs recommended in Phase 1. Figure 1 summarizes the activities, showing initial states in green, interim states in yellow, and products in blue.

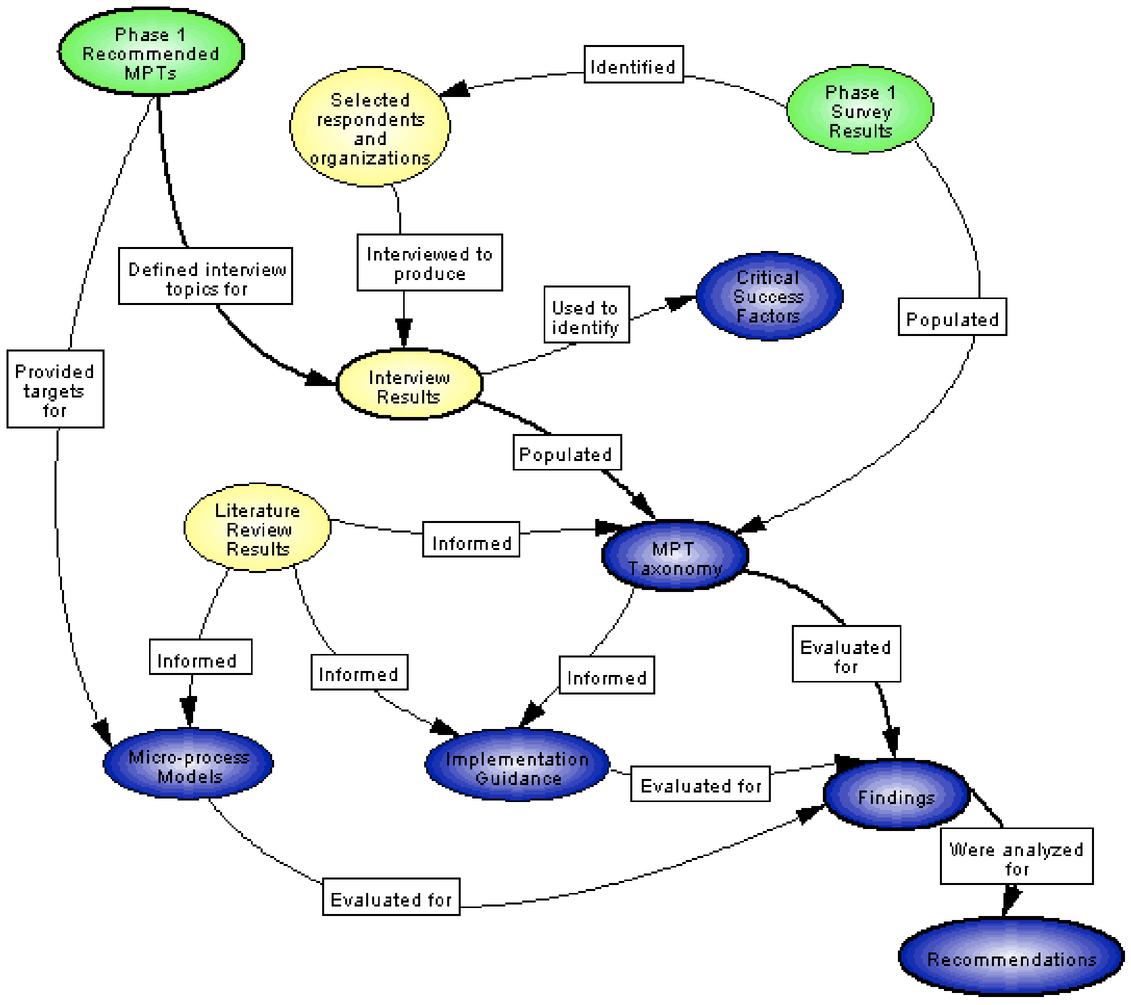


Figure 1. Systemigram of Phase II activities

The results of this work are directly relevant to the challenges currently being faced in the sponsor environment, and support sponsor-directed future research.

- **Evaluation and Support of existing programs.** The MPT taxonomy (see 3.3, 4.3, 5.1, and Appendix B) has articulated key challenges that must be addressed in order to deliver agility in system engineering, along with MPTs that have been helpful in some contexts and important gaps still remaining to be addressed. The taxonomy can be used to evaluate whether teams are cognizant of, and addressing, the key challenges, as well as to assess whether MPTs are currently being applied with the necessary level of rigor.

- **Implementation guidance for existing programs.** The three recommended MPTs highlight existing, well-defined practices that can help achieve beneficial system engineering results under relevant constraints. The implementation packages (see 4.5 and Appendix D) developed for each of these MPTs are designed to help teams interested in adopting them to make an informed decision about whether adoption will yield worthwhile benefits, and to understand the rigor that is necessary in order to yield the benefits.

- **Systems Engineering Transformation (SET).** The work directly supports the Systems Engineering Transformation roadmap development by providing a baseline of SE practice in the sponsor environment and identifying gaps identified to understand the types of MPTs that need to be developed and evaluated (SERC-2009-02, September 2009). The critical success factors (see 4.2) can be used in correlating existing, new, and proposed MPTs and their target effects with project success in the rapid development and innovation environment. Focusing on MPTs that address gaps and also support the critical success factors may provide greater benefit in the near-term.

- **Research in difficult to access environments.** The Little-JIL micro-process modeling (see 3.2, 4.4, 5.2, and Appendix C) has demonstrated the feasibility of this approach for matching candidate MPTs to teams working in various contexts. These results show that this is likely to be an effective approach for filtering both newly proposed and existing MPTs, in order to make recommendations about which are likely to be the best fit for various teams. The success of the modeling and analysis also indicate the usefulness of a test bed to exercise new ideas before piloting, elevating the probability of pilot success.

2 Introduction

The Systems Engineering Research Center (SERC) Evaluation of Systems Engineering (SE) Methods, Processes and Tools (MPTs) research effort was initiated to provide a broad sense of the availability or absence of useful SE MPTs, particularly in a fast-paced, network-enabled, emergent development environment.

2.1 Definitions

An MPT is a systems engineering technique that fits into one of the following categories:

Method (M) – A collection of inter-related processes, practices, artifacts, agents, resources and tools. A method is essentially a "recipe." It can be thought of as the application of inter-related processes, practices and tools wherein different agents use resources to create and apply artifacts to a class of problems.

Process (P) – A logical sequence of steps (tasks) intended to achieve an objective. The objective achieved may be abstract (e.g. “negotiate among multiple stakeholders”) and/or a composite of multiple individual goals (e.g. “Deliver a fixed-date, variable-scope system”). Performance of a step is often the responsibility of an agent, which may be a human, a device, or a software system. Performing the step may consume resources and require access to various kinds of artifacts in order to execute. Execution of a step will generally produce more artifacts. The structure of a process enables several levels of aggregation (i.e. sub-processes) to allow understanding and analysis of the process at multiple levels of abstraction in support of decision-making.

Tool (T) – A tool automates or partially automates one or more steps within a process and thereby enhances process performance efficiency.

A **useful** MPT is defined as one that is:

- *Relevant to the application environment:* applicable to some subset of systems within the target environment.
- *Repeatable:* sufficiently well defined that implementation is possible in a different context.
- *Likely to have significant impact:* can materially improve systems engineering practice in the application environment.

A **viable** MPT is successfully implementable in the target organization given appropriate and reasonable tailoring.

2.2 Approach

This task follows directly from the results of the initial MPT work.[1] The three MPTs recommended in the previous phase (Scrum, rapid prototyping, and continuous integration) were the focus of most of the research.

Further investigation of MPTs was performed through follow-up interviews and additional research. Based on the analysis of the interviews, a set of critical success factors for development in rapid, innovative environments was identified. The guidance used in the interviews is found in Appendix A.

A taxonomy for MPTs was developed based on previously gathered information augmented by the interviews. The taxonomy was used to describe the three MPTs recommended in the previous phase (Scrum, rapid prototyping, and continuous integration). The taxonomy used themes defined from previous work and concepts from micro-process definitions to establish and populate descriptions of MPTs. These descriptions were shown to be useful for both implementing and validating the use of MPTs. The taxonomy for the three MPTs is found in Appendix B.

To investigate the possible benefits of micro-process modeling in analyzing MPTs, the Little-JIL micro-process modeling tool was used to develop models of Scrum. Analyses were successfully performed using the models, including single point of failure and finite state verification techniques. The results of the analysis were sufficient to validate the use of such modeling techniques in the future. The models and analysis results are found in Appendix C.

Implementation guidance for the three MPTs was developed based on the taxonomy and the generic understanding of the environment defined in Phase I. The completed implementation guidance packages are found in Appendix D.

Section 3 provides more detail on the actual execution of the approaches.

SERC organizations involved in the data collection and analysis include the Fraunhofer Center for Experimental Software Engineering, Missouri University of Science and Technology, Stevens Institute of Technology, the University of Alabama in Huntsville, the University of Massachusetts at Amherst, and the University of Southern California.

[1] Carrigy et al, “Evaluation of Systems Engineering Methods, Processes and Tools on Department of Defense and Intelligence Community Programs: Phase 1 Final Technical Report,” Systems Engineering Research Center, SERC-2009-02, September 2009.

3 Methods, Assumptions and Procedures

A number of research methods were used to gather and analyze information for this task. Follow-up interviews with individuals who had responded to the industry survey played a major role. Additional interviews were conducted with organizations with strong similarity to the sponsor environment and success in rapid response development. Formal process models of proposed MPTs were built and analyzed. The taxonomy from Phase 1 was extended and applied to the three recommended MPTs, and the insights were applied directly to developing implementation guidance.

3.1 Interviews

Phone interviews were held with selected respondents to the Phase I industry survey and with representatives of several commercial organizations that had specific experience with the sponsor or whose particular business practices seemed highly applicable to the sponsor's environment. The primary intent of these interviews was to gain additional insight into how organizations with some similarity to the sponsor environment were implementing the three primary MPTs (Scrum, Rapid Prototyping, and Continuous Integration). Additionally, questions were asked to better understand the types of projects individual respondents were working on, to identify other MPTs seen to be successful, and to gain additional insight into gaps in current MPTs.

Interviews were also held with representatives of organizations involved in both large software intensive system development as well as rapid response system development. These interviews were primarily used to derive critical success factors.

The guidance and questions used in these interviews can be found in Appendix A. Relevant information obtained through these interviews was integrated into the taxonomies for each MTP. The information gathered was also analyzed to identify critical success factors.

3.2 Micro-process Modeling

Process modeling is one way to study and analyze how well MPTs fit within an organization's workflow. As an exemplar for this approach, the team modeled the recommended method Scrum in Little-JIL, a micro-process modeling language developed at the University of Massachusetts, Amherst. The language is based on formal semantics drawn from finite state machine representations, which allows a broad variety of analysis techniques to be applied. Once constructed, the models demonstrated their usefulness through two types of analysis activities: fault tree analysis and finite state verification. The models and the analysis are provided in Appendix C.

3.3 Taxonomy Development

A comprehensive taxonomy was developed to relate the key challenges involved in agile system engineering, specific MPTs to address those challenges, and the specific component parts of each MPT. To minimize subjectivity, this taxonomy was based on the responses of practitioners from the Phase I survey. Survey responses were examined to identify how each MPT recommended by industry and government practitioners solves a given problem, to determine important underlying approaches. For example, a variety of requirements challenges may be addressed through effective communication techniques, leveraging expert personnel, or modeling. These general approaches were termed MPT "themes." By understanding the themes underlying common MPTs, new MPTs can be created and current MPTs adapted to operate within the sponsor environment while still retaining the aspects that make them successful in addressing the challenge areas.

3.3.1 Qualitative analysis of survey responses

A multi-step process of qualitative analysis was performed to identify common themes of *how* the MPTs mentioned in the survey responses addressed the various challenges. Each step involved *open coding*: the process of identifying the categories in qualitative data and the properties of those categories.[2] First, the survey responses were examined to identify unique MPTs for each challenge area. In total, more than 200 unique MPTs were identified. Many MPTs (e.g. rapid prototyping) were suggested for multiple challenge areas.

Second, the unique MPTs were grouped into categories of MPT themes. The themes represent a strategy of *how* the MPTs address a particular challenge (e.g. prioritization methods, personnel changes, change management). The themes and unique MPTs grouped under them were reviewed and approved by a team of three researchers. More than 30 themes were identified, and some themes (e.g. direct stakeholder communication) appeared for multiple challenge areas. Table 1 summarizes the number of MPTs and MPT themes identified for each challenge area.

Table 1. Counts of MPTs and MPT themes by challenge area

Challenge area	Specific problems identified	MPTs identified	MPT themes
Requirements	22	89	17

[2] A. L. Strauss and J. M. Corbin, *Basics of Qualitative Research: Techniques and Procedures of Developing Grounded Theory*, Second Ed., Sage Publications, Thousand Oaks, CA, 1998.

Stakeholder issues	14	71	13
Sustainment	9	41	14
Integration	12	50	15

3.3.2 Example results – Codes for Stakeholder challenge area

The coding results of MPT themes for the stakeholder challenge area are depicted in Figure 1. Direct stakeholder communication was the most commonly identified theme among the MPTs in this challenge area. MPTs in this theme include “meet the customer,” “participate in daily Scrum meeting,” “frequent stakeholder meetings,” and more. The themes are a useful way of categorizing the survey responses, especially those that do not mention a specific practice. For example, while “meet the customer” is not a specific technique, it is indicative of a general approach to meeting the challenge. Clustering MPTs into themes allows further analysis to be done to identify common operational concepts.

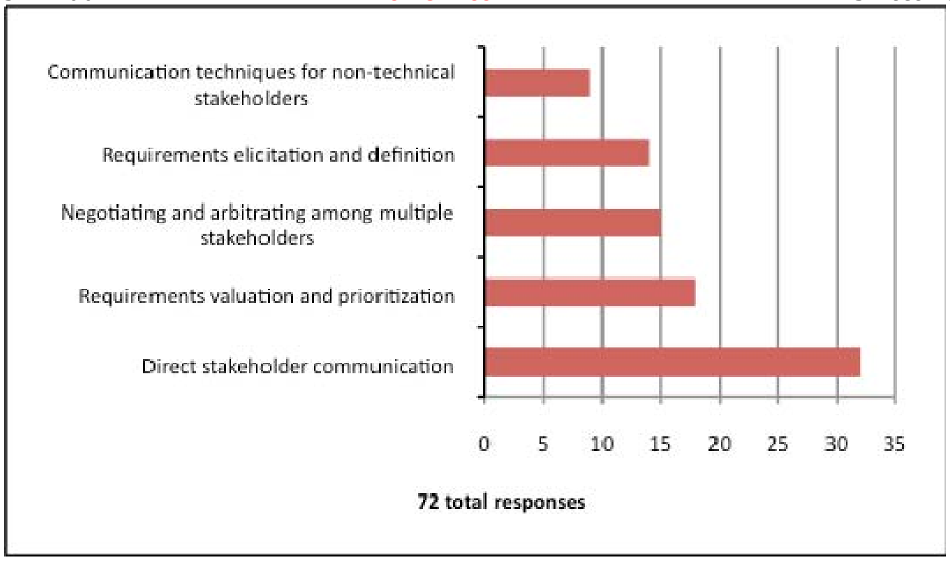


Figure 2. Most common themes for the Stakeholder challenge area

3.3.3 Mapping concepts to practices

In order to utilize the MPT themes for future scientific inquiry and to direct industry adoption of MPT themes, we have created MPT bridge diagrams. An MPT bridge diagram is comprised of three key elements: 1) the MPT theme; 2) the conceptual elements of the theme; and 3) a taxonomy of methods, processes and tools related to the theme. Figure 3 depicts the bridge diagram for the "iterative development or frequent delivery" theme. Rapid prototyping, a recommended MPT, is highlighted in green, and other MPTs recommended in the survey under this theme are outlined in an olive tone.

The conceptual elements of an MPT are the constituent parts which define the theme. In Figure 3, a decision/arbitration process, obtaining stakeholder communication for input and feedback, and the valuation of requirements are the hallmarks of the theme. All three of these elements must be addressed or instantiated by a concrete practice in order to perform "negotiation/prioritization among stakeholders." The taxonomy on the right hand side lists the MPTs under this theme and links them to the conceptual element they instantiate.

The bridge diagram serves three purposes. First, going from left to right, an organization adopting an approach to solving a challenge (i.e. adopting an MPT theme) can identify which MPTs are available to instantiate the conceptual elements of that theme. Second, going from right to left, an organization can assess whether or not their existing practices match the conceptual elements for a theme. If not, then there is some risk the theme is not properly instantiated and perhaps wasteful. Finally, the diagrams can help illustrate gaps between the conceptual elements of a theme and the state of the practice, i.e. where new MPTs need to be created or adapted in order to address a particular challenge area met by a theme.

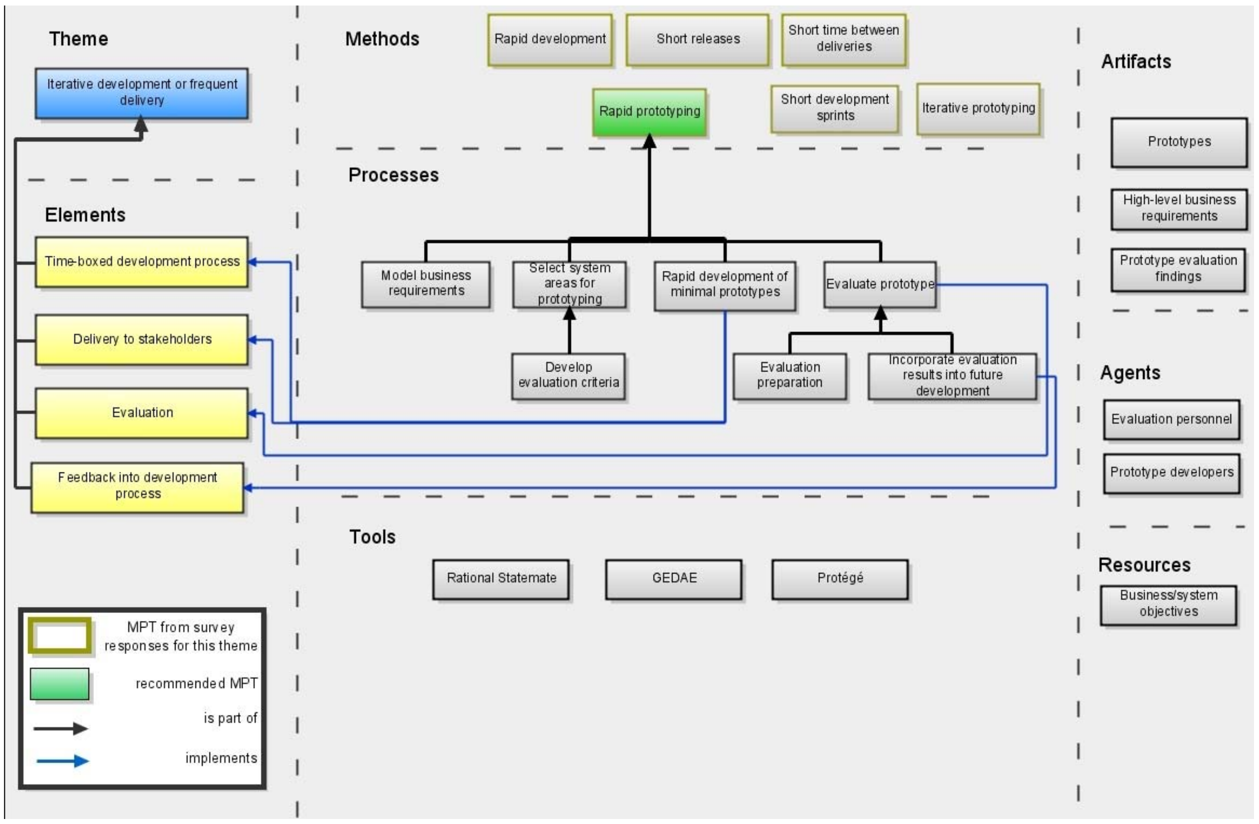

Theme
Iterative development or frequent delivery
Elements
Time-boxed development process
Delivery to stakeholders
Evaluation
Feedback into development process
Methods
Rapid development
Short releases
Short time between deliveries
Rapid prototyping
Short development sprints
Iterative prototyping
Processes
Model business requirements
Select system areas for prototyping
Rapid development of minimal prototypes
Evaluate prototype
Develop evaluation criteria
Evaluation preparation
Incorporate evaluation results into future development
Tools
Rational Statemate
GEDAE
Protégé
Artifacts
Prototypes
High-level business requirements
Prototype evaluation findings
Agents
Evaluation personnel
Prototype developers
Resources
Business/system objectives
MPT from survey responses for this theme
recommended MPT
is part of
implements

Figure 3. Bridge Diagram for "Iterative Development or Frequent Delivery" theme

This page intentionally left blank

4 Results and Conclusions

The research team collected and analyzed data from the sources described in Section 3. The results and conclusions as described in this section were the basis for the recommendations in Section 5.

As described in the Summary (Section 1), these results are directly relevant to the challenges currently being faced in the sponsor environment, and support sponsor-directed future research. Particularly,

- **Evaluation and Support of existing programs.** The MPT taxonomy has articulated key challenges that must be addressed in order to deliver agility in system engineering, along with MPTs that have been helpful in some contexts and important gaps still remaining to be addressed. The taxonomy can be used to evaluate whether teams are cognizant of, and addressing, the key challenges, as well as to assess whether MPTs are currently being applied with the necessary level of rigor.

- **Implementation guidance for existing programs.** The three recommended MPTs highlight existing, well-defined practices that can help achieve beneficial system engineering results under relevant constraints. The "packages" for each of these MPTs are designed to help teams interested in adopting them to make an informed decision about whether adoption will yield worthwhile benefits, and to understand the rigor that is necessary in order to yield the benefits.

- **Systems Engineering Transformation (SET).** The work directly supports the Systems Engineering Transformation roadmap development by providing a baseline of SE practice in the sponsor environment and identifying gaps identified to understand the types of MPTs that need to be developed and evaluated. The critical success factors can be used in correlating existing, new, and proposed MPTs and their target effects with project success in the rapid development and innovation environment. Focusing on MPTs that address gaps and also support the critical success factors may provide greater benefit in the near-term.

- **Research in difficult to access environments.** The Little-JIL micro-process modeling has demonstrated the feasibility of this approach for matching candidate MPTs to teams working in various contexts. These results show that this is likely to be an effective approach for filtering both newly proposed and existing MPTs, in order to make recommendations about which are likely to be the best fit for various teams. The success of the modeling and analysis also indicate the usefulness of a test bed to exercise new ideas before piloting, elevating the probability of pilot success.

The following paragraphs discuss the results of the individual activities.

4.1 FOLLOW-UP INTERVIEWS

A portion of the previously conducted industry survey contained fields for indicating willingness to join a community of interest and providing contact information. Of the population who provided contact information, candidates for interviews were selected based upon closeness of fit to the sponsor's environment. Only survey respondents of Good or Excellent fit to the sponsor's environment as described in the interim progress report were contacted for follow up interviews. As an additional filter, only respondents who had answered at least half of the discussion questions were considered. To date, for this group of 26 potential interviewees, 7 responded affirmatively, 3 declined, and the remainder did not respond to the request. Only 5 participants could be scheduled within the remaining timeline of this task. Interviews were conducted with these representatives of 5 different companies in different industries.

The following list summarizes the projects discussed for each of these industries:

- Aerospace (Respondent A) – Multiple complex development efforts with safety as the primary stakeholder value, followed by reliability and cost. Development time can be decades, but still involve frequent requirements changes. The development environment tends to involve stovepipes within and between programs and concurrent execution, often with blinders on. Interoperability of multiple subsystems exists.

- Defense (Respondent D) – Multiple secret and top-secret projects with 18 to 48 month delivery cycles of 100-500 KLOC software with complex algorithms and volatile requirements. Systems must interface with other systems for sharing of unclassified data, while keeping classified data inaccessible, including anti-tamper protection on chips. Schedule was the primary driver, projects were also evaluated by operational capability, accuracy, integrity, availability, continuity, and cost.

- Gaming (Respondent G) – Developing the system to run an entire floor of games provided by multiple vendors. This schedule driven project was a 1 year development effort with 200 employees and 20 teams. Interim and final products were delivered. The project required interfacing with other systems while maintaining information security (both for proprietary information and for personal information from end users). 250-300 features were included. Cost was not considered a priority.

- Healthcare IT (Respondent H) – This environment must show rapid progress while adhering to statutory requirements, maintaining security of information (with monetary penalties for breaches), and managing frequent requirements changes. The total development cycle is 18 months with quarterly delivery, and 200 story points per 2 week cycle (5000 function points per product).

- Transportation (Respondent T) – This program developed and implemented an SE process guidebook for Intelligent Transportation Systems in a DOT context. The project included software development as well as integration with an environment requiring expertise in physical infrastructure development. Projects may be low, medium, or high in complexity, and cost is not necessarily an indication of complexity. Schedule is in terms of years.

The following highlights did not apply directly to the MPTs in the taxonomy, but are still of interest:

- Many organizations face the same problems (e.g. dynamic requirements) which are not addressed by current MPTs. New ideas in SE are needed to address them.

- MPTs cannot be replacements for thinking and communicating because those are both key to good engineering and design. The expectation of tools to provide answers leads to less thinking and validation that can cause issues downstream.

- Having the right people with the right training is essential.

Relevant information obtained through these interviews has been integrated in the sections below.

4.1.1 Scrum

Three of five respondents used Scrum. Two considered Scrum beneficial, while the third was not certain if Scrum was beneficial. Sprints ranged from two to four weeks, with daily stand-ups for two-week sprints and twice weekly stand-ups for the four week sprints. Communication was a challenge cited by all respondents.

Respondent H's implementation of Scrum uses two weekly meetings (instead of daily stand-up meetings). Communicating problems with the team during these meetings was cited as an issue. They have ~200 story points per two week sprint. This has led to specific velocity expectations from the customer, and the respondent would recommend using "percentage of work left" as a measurement to the customer instead of velocity (# story points completed/sprint).

Respondent G used some Scrum practices, under the name "iterative development." The greatest challenge was running many parallel teams. Each team held daily stand-ups. People (including team leads) were on multiple teams and just scheduling daily stand-ups was an issue. The project involved both new development and the integration of legacy components. Development cycles were four weeks in length with an interim and final product release.

The requirements were frequently changing in terms of what iteration features were implemented. As iterations were accomplished, features that had not been met were shifted to later iterations. The shift wasn't necessarily to the next iteration. Reallocations were managed through a Change Control Board.

This project was a successful implementation of Scrum. Deadlines were met, but the interviewee indicated that an earlier focus on performance and more effective use of performance modeling would have improved the process.

Respondent D has used Scrum on less than 10% of programs. The program manager represents the customer on these programs, and the requirements and product backlog are formally documented as part of EVM. Product backlog was used as a measure and updated after each daily meeting. It was unclear if Scrum improved success of the programs.

System architecture was found to be essential for daily meeting. Self-management was cited as a struggle. The ineffectiveness of scrum-of-scrum for coordinating multiple teams meant that Scrum did not scale well beyond 10 people.

4.1.2 Rapid Prototyping

Four of five participants used Rapid Prototyping.

One used prototypes frequently and integrated successful prototypes into the developed system. The expectation was that prototypes would be "designed for reality." In cases where prototypes were not integrated, lessons were learned, but time was wasted.

Three used rapid prototyping to aid in communicating with the customer and problem solving, but developed as a separate effort (with hardware prototypes potentially reserved as back-up systems). One cited budgetary constraints as a limitation. Another indicated that sometimes rapid prototyping is a formal part of a program, but often it is an internally-funded activity to accelerate technology maturity. Ensuring that prototypes are discarded and not carried forward was considered the greatest challenge. Rapid prototyping was considered beneficial in terms of reducing rework during the design & development which reduces cost and schedule risk.

Respondent H uses limited Rapid Prototyping as a means of communication with customer.

Respondent A uses Rapid Prototyping as a means of problem solving early on, but the budget rarely allows for much rapid prototyping.

Respondent G used Rapid Prototyping throughout the development. Whenever possible, prototypes were integrated into the system. The expectation was that prototypes would be "designed for reality." In cases where prototypes were not integrated, lessons were learned, but time was wasted.

Respondent D typically uses rapid prototyping. Sometimes this is a formal part of a program, but often it is an internally funded activity to accelerate technology maturity.

Ensuring that prototypes are discarded and not carried forward into E&MD was considered the greatest challenge. Rapid prototyping has been beneficial in terms of reducing rework during the design & development which reduces cost and schedule risk.

4.1.3 Continuous Integration

Three of five respondents used continuous integration (CI). The two remaining respondents have considerable hardware integration that limits the feasibility of CI. All participants using CI found that automation of some processes was beneficial.

Respondent H uses CI with nightly builds with “very positive results.”

Respondent A works with large/complex hardware development that does not lend itself to continuous integration. Integration occurs on a milestone basis.

Respondent G started using CI during the 3rd iteration. Initially, integration was all manual, and trying to implement CI on a schedule driven project was a “nightmare.” Once CI was integrated, it was beneficial, but starting CI sooner would have lead to greater success.

Respondent D only uses CI on projects using Scrum. Textbook CI is used on these projects. Automated static analysis is seen as a benefit, but the respondent stated that they have no quantitative data to prove this.

The typical project involves manual builds initiated by the project engineer. Automated integration testing is used, but it is also manually initiated.

4.1.4 Other MPTs Mentioned

Respondents also discussed or alluded to other MPTs. Change Control Boards were recommended for managing requirements reallocation. Two respondents discussed the benefits of pilot programs with customers prior to final release. Piloting allows users to provide feedback while the system is still in development and may not have full functionality. Also, implementation issues not captured during testing may be highlighted through this type of release. In both cases, only a small number of users were involved in pilots. Separation of sustainment and development with a distinction between enhancing code (sustainment) and adding new functionality (development) was suggested.

While none of the interviews were aimed at evaluating software tools, several were mentioned, including Code Test, ClearCase, DOORS, Klockwork, LDRA and Test Real Time.

4.1.5 Gaps

There is no doubt that practitioners are concerned about methods, processes and tools that do not meet their day to day needs and they seemed unable to find answers in many areas. The following are gaps they mentioned:

- Tool for creating incremental requirements.
- Tools that help identify code and algorithms that are inherently sequential (that can't be split into pieces running in parallel on independent cores) or that are susceptible to parallelism are needed.
- Static analysis tools capable of operating across multiple cores with low false positive and low false negative rates are needed – especially for detecting race, deadlock, livelock conditions.
- A real-time operating system that can dynamically assign tasks among cores may be needed – possibly a hypervisor that coordinates the migration of tasks from one core to another in a multitasking environment.
- Tools for visualization and analysis to support "around the table" problem solving.
- New ideas in SE. Many organizations face the same problems (e.g. dynamic requirements), which are not addressed by current MPTs.
- Critical Thinking. MPTs cannot be replacements for thinking and communicating because those are both key to good engineering and design. The expectation of tools to provide answers leads to less thinking and validation that can cause issues downstream.

4.2 Critical Success Factor Analysis

Based on the analysis of a set of specially arranged organizational interviews, critical success factors were identified that enable successful development in a rapidly changing environment, particularly one focused on innovation. Most of the organizations interviewed are involved in both traditional and rapid response system development. The team interviewed:

- The Aerospace Corporation's Concept Design Center (www.aero.org)
- Institute for Creative Technologies, University of Southern California (http://ict.usc.edu/about)
- Lockheed Martin Corporation's Skunk Works (Skunk Works Today | Lockheed Martin)

- Northrop Grumman's Futures Lab, joint venture with Applied Minds (http://appliedminds.com/)
- Commercial Rapid-Development Company (requested anonymity).

Because all of the respondents considered the information discussed as proprietary, actual responses captured from each interview are not provided. Rather, this section of the report describes critical success factors that were common at several sites, if not all sites.

4.2.1 Early Concept Exploration and Feasibility Assessment

All of the organizations that provided inputs indicated the importance of early concept exploration and feasibility assessment that often required considerable modeling and prototyping. The level of modeling and prototyping varied, typically based upon perceived risks of the technical approach or the technologies to be integrated into the solution. In order to encourage innovation, organizations think that it is important to establish a supportive culture and environment.

4.2.1.1 Investment in Innovation Environment

Several organizations pointed out the importance in investing in innovation and technology maturation ahead of an identified need, especially when customers may need rapid responses to changing needs, missions, and threats. Innovation is very difficult to achieve in stressful situations. Starting with a clean sheet of paper and designing a solution quickly may produce a useful solution given the right engineering expertise, but it will probably not reach the level of innovation.

To enable innovation, organizations:

- **Include Responsible Play:** Organize work to include responsible play with new concepts and ideas in a supported lab environment
- **Focus on Team Rewards:** Set up a collaborative environment that rewards team work rather than individual work. This leads to sharing and collaborating without fear that their personal rewards (e.g., promotions, raises, bonuses) will suffer if someone else gets the credit.
- **Use Both Science and Art:** Learn to balance engineering focus between science and art.
- **Make it OK to Fail:** It is often through failures that people learn and adapt ideas.
- **Leapfrog:** It should also be not-OK to not-fail. Keep teams from trying for 20% improvements; go for at least a factor of 2.

- **Multi-sourcing:** If it's OK to fail, you want to have several teams trying different approaches. This also stimulates the competitive juices, often even within an organization. Some commercial companies have 3 design shops that compete for the next version of their product.

4.2.1.2 Root Cause Analysis of Customer Problem

Spend time investigating the root cause of a customer's problem. Sometimes the best solutions focus on eliminating the root cause of the problem rather than developing something to deal with the problem.

4.2.1.3 Reality Confrontation

Early prototypes are invaluable in both understanding the requirements through iterative feedback from customer and understanding the capabilities and limits of new technologies or existing technologies used in new ways. Much is learned from taking a design on paper and translating it into a prototype that designers, customers, and potential users can interact with. Have a working prototype on Day 2, and have real users ready to exercise and comment on it. A combination with Leapfrogging is to do a factor-of-1.5 solution, get some quick experience with it, and then try for a factor-of 4 solution. If you have to back off to a factor-of-3, you're still ahead.

4.2.1.4 Customer or Sponsor Commitment and Participation

For those cases where efforts are applied to a specific customer need, customer/sponsor commitment and participation are extremely important. In fact, at some sites, if the customer/sponsor does not provide the needed level of commitment and participation in developing and assessing the feasibility of the requested solution, work is deferred. The customer/sponsor participation is required to provide insights into the requirements/user needs as well as to interact with models and prototypes to give feedback to the concept developers. Note that innovative design may have no identified customer or sponsor. For example, when the organization is attempting to develop a breakthrough commercial product for a totally new market, they may rely on market surveys and trends rather that a specific customer or sponsor.

4.2.2 Value-Adding Tools with which Users have Experience

Tools are required to succeed in this environment. However, the tools must be the right (value-adding) tools and the users must be experienced with those tools. The wrong tool or the right tool with no team expertise is not of value. For those organizations that periodically tap their key corporate resources (i.e. super-stars) to work on special innovative, rapid response projects or to conduct feasibility assessments of concept designs, it is important that the project work environment include the tools that those team members use in their day-to-day work. Another key theme is that tools don't need to be the best or the most sophisticated. Sometimes it is the simple, stable tools that work best.

4.2.3 The Right People

Most agree that you can have the best tools and the best processes, but without the best people success is difficult at best. To achieve the desired results in an innovative, rapid-development environment, organizations need to enable the best to achieve the desired task.

4.2.3.1 Empower the Best

For the rapid response and up-front innovation, prototyping, and feasibility assessment work, organizations typically focus on their super-stars and experts in the domain(s) of interest. (Some experiences indicate that super-stars can be as much as 10 times more productive than the average performers.) These people work together as a small, lean team, collaborating almost continuously and developing frequent iterations and refinements of concepts until the desired solution is identified and adequately understood. Managers of these teams typically have full authority and responsibility for them and the technical members are empowered to make the technical decisions. Because of the relatively small size of many of these teams, the project organization is often flat. For larger projects (e.g., new aircraft design and development), teams are still relatively small when compared to the traditional team size, but there are typically not enough super-stars to fully staff the project. However, some super-stars mixed with committed and very experienced team members are still the norm.

4.2.3.2 Enable Holistic Concurrency

Have experts on tap who cover the key fielding considerations and their tradeoffs (for example performance, reliability, usability, producibility, evolvability, cost), who participate concurrently rather than sequentially, and who pro-actively keep up with the state of the art in their domains.

4.2.3.3 Identify a Keeper of the Holy Vision:

The strongest successes come when the team has someone with enough range of expertise and experience to understand and synthesize the components of solutions, and to bring the right people together when problems come up (e.g., Kelly Johnson's principles for Boeing's Skunk Works).

4.2.4 Supportive Work Environment

Whether or not the work is classified or proprietary, the innovative, rapid development teams tend to work in their own large, relatively unstructured open space (sometimes with cubicles) to encourage collaboration and experimentation. When the same key people are being used frequently for intense rapid-response projects, it is important for the organization to provide additional resources and rewards that will help these people with their outside lives (e.g., family, external commitments). If people on the teams are overly stressed, innovation and creativity will suffer. They could easily end up reverting to a 9-to-5 mode that is counter-productive to the rapid-response goals.

4.3 Population and Usefulness of Taxonomy

Many distinct MPTs were recommended by survey respondents for addressing the sponsor's challenge areas. The recommendations included specific, well-defined MPTs (e.g. Scrum, Incremental Commitment model, Pugh value analysis) and abstract suggestions (prototype, frequent demonstration to the customer). The total number of unique MPTs and MPT themes for each sponsor challenge area are shown in Table 2. Some MPT themes and MPTs appear in multiple challenge areas.

Table 2. Total number of identified MPTs and MPT themes

Challenge area	**MPTs**	**MPT themes**
Requirements – Changing requirements priorities and/or emerging requirements	17	76
Stakeholders – Obtaining useful stakeholder input and dealing with conflicting stakeholder requirements	14	56
Sustainment – Conflicts between developing new capabilities and supporting the currently released system	14	37
Integration/interoperability – Integrating independently evolving components into a larger system	15	48

Discussions with the sponsor suggested two principal objectives for using the taxonomy in the sponsor organization:

1. To provide a list of potential techniques for addressing a development and the process engineering required to implement them (i.e. a methodology shift, transitioning to a new process, acquiring a new tool);

2. To support assessment of implemented MPTs by identifying the critical elements that must be present in the development process.

The taxonomy shares definitions with elements of the micro-process modeling in Little-JIL. The taxonomy has both a textual and graphical representation. Because of the large number of potential MPTs, the textual or graphic representations of the taxonomy are depicted for one Method or Process each. That is, the taxonomy applied to a Method or Process will show the constituent (sub-)processes, Tools, Agents, Artifacts, and Resources for that MPT (see Appendix B for definitions of Agents, Artifacts and

Resources). In future iterations of the taxonomy, a tool will be necessary to capture all of the MPT taxonomy information in a single, interactive location.

The taxonomy developed is presented in Table 3. The taxonomy contents, as populated to date, are found in Appendix B.

Table 3. MPT Taxonomy layout with truncated example

MPT	*Continuous Integration*
Theme	Frequent/continuous integration
Methods	Continuous integration
Processes	Integration process; Commit to the source repository; Automated build; Self-testing build; Commit to the mainline daily; Build the mainline on an integration machine; Build time optimization; Communication process; Automated deployment process
Artifacts	Source code; test code; build results; automated testing results; deployment scripts
Agents	Integration engineer
Resources	Clone of the production environment for testing; dedicated integration build machine; single source repository
Tools	Continuous Integration – AccuRev, Anthill Pro, Apache Continuum, Apache Gump, Automated Build Studio, Bamboo, CABIE, FinalBuilder, Hudson, Parabuild, TeamCity, Team Foundation Server, Java (Ant, Ruby, etc), .NET (Nant, MSBuild, etc), XUnit (Java – JUnit), FIT, Selenium, Sahi, Watir, FITnesse, Capistrano

The development and tailoring of the taxonomy was a significant milestone. Anecdotal evidence abounds of teams who claim they are applying a particular MPT, but have tailored the process in such a way that key steps are missing, thus losing the beneficial effects of the process. Conversely, MPTs that cannot be tailored to specific constraints found on different teams will not be widely useful. Thus the primary goal of the taxonomy was to describe MPTs in sufficient detail such that their key components (processes, methods, artifacts, and resources) were identified and described, but without imposing so much detail that the taxonomy would not be useful to system engineering teams. Describing the MPTs using the taxonomy requires the user to research the minimum set of required components, how those components should be performed, and what benefits those components provide – that is, why each of those components was truly required in order for teams to see the expected benefits of using the MPT.

4.4 Effectiveness of Micro-process Modeling

The modeling and analysis of Scrum using the Little-JIL language illustrated three significant ways such an activity can support MPT research:

1. Clear understanding and description of MPT relationships

2. Capability for formal analysis of MPT components at any level

3. Capability to identify impact of changes to the MPT at any level

The act of defining an MPT using Little-JIL clarifies the relations among the processes, subprocesses, tools, agents, artifacts, and resources that comprise the method. It is easy to identify how various processes comprise the method, the ways subprocesses comprise the processes, how processes and subprocesses generate and consume artifacts and products, the identities of the various agents and their responsibilities, and the existence or absence of mechanisms to coordinate concurrent access to key artifacts.

While any clear graphical notation allows consideration of these relations, using a rigorously defined process language allows researchers and practitioners to evaluate, discuss and improve the model based on reasoning supported by the rigorous definition. The Little-JIL process definition language semantics are based on finite state machines to provide the precise semantic meanings of each of the language's components.

The underlying rigor also serves as the basis for powerful analyses to offer greater insights into such processes, to support integration with other methods, and to detect defects and vulnerabilities. The Little-JIL Scrum method definition was used successfully in such analyses. A fault tree was generated automatically and used to identify a single point of failure in the process. This facilitated the removal of that process vulnerability, as verified by analysis of a second automatically generated fault tree. A second example showed how finite-state verification can be applied to a process definition in order to identify process defects. One tool (PROPEL) was used to define a specification of desired behavior, and another tool (FLAVERS) was used to scan a graph that had been automatically generated from a Little-JIL process. FLAVERS identified a process path whose execution would violate the PROPEL statement of desired behavior, and further analysis suggested how the defect might be removed. These two demonstrations underscored the value of a process defined using a rigorous language, and the power of tools designed to exploit the rigor of the language.

Finally, by having an executable model that supports both defect analysis and measurement, changes to the MPT can be implemented and their impact can be evaluated before in vivo piloting. This provides the basis for a broad spectrum of experimentation, ranging from process refactoring all the way to totally new approaches.

4.5 MPT Implementation Guidance

The implementation guidance for each of the MPTs recommended in Phase I is found in Appendix C. Based on previous experiences with tech transfer, such packages need to go beyond the textbook definition of an MPT—if the MPT would work as is, it would likely have already been adopted. Rather, the packages need to include practical guidance related to how the MPT needs to be tailored for the real world. On the other hand, a common problem we have found is that teams may inadvertently tailor away some of the beneficial aspects of an MPT, so that what they claim to be doing does not bear a substantial resemblance to the real practice, and may not be as likely to be effective. For this reason, the package also includes a section that describes the minimum level of rigor that needs to be found in actual application in order to find the promised benefits. This rigor is specified using the elements of the taxonomy developed in this work, namely sub-processes, methods, resources, tools, and artifacts.

Specifically, each package contains information on:

- Why it is recommended: To help teams determine whether the MPT is a good fit for their environment and constraints, a description is included of the practical challenges it helps to address.

- Contexts where it is suitable: Not every MPT is a good fit for every team. Information is provided as to what environmental aspects can facilitate or enable the potential benefits of the MPT, and which make it more difficult or entirely unsuitable for application.

- Known impacts on cost, quality, and schedule: Every practice has tradeoffs. Experience reports and other measures from our interviews, survey, and literature review are summarized to present an overview of the costs and benefits of each MPT.

- How to use it / monitor it: To avoid the problem of teams tailoring away necessary rigor, the basic processes, methods, artifacts, or resources that need to be in place are described. On any team which is effectively applying the MPT, these aspects should be visible. References to the appropriate taxonomy sections of this report, where additional detail can be found about these component pieces, are also provided.

- Useful resources: Teams that decide to adopt the MPT may find items on this list helpful for putting it into practice. These items include concise overviews, reusable resources, and helpful tools.

4.6 Community Building

Although the research team had wide-ranging expertise in the area of agile system engineering, no single team can have sufficient expertise across all kinds of development contexts. This turned out to be responsible for one of the significant results of this effort. The tight constraints and challenges of the sponsor’s environment led the team to work with practitioners facing similar contexts. Through them, verification of the recommendations against practical experience was possible, and an understanding of whether and how the recommended MPTs have been successfully tailored to this context. The identification and engagement with this set of practitioners, first through the industry survey and later through more comprehensive follow-up interviews, were key to the research findings. Beyond providing useful input to the research products, these practitioners will form the basis of a community of system engineers interested in similar ideas about infusing agility into the development process. There is no doubt that they will make positive contributions to the future work as well.

This page intentionally left blank

5 Recommendations

The following recommendations represent the consensus of the research team. Other recommendations were considered. Succeeding tasks will refine and augment those provided.

5.1 Recommendations on Taxonomy and Implementation Guidance

The MPT taxonomy structure has been iteratively refined and now is believed to be robust and a suitable foundation from which to categorize MPTs to support systems engineering and assessment in the sponsor environment. At present, the taxonomy has been applied to the three recommended MPTs (Scrum, continuous integration, rapid prototyping), though more MPTs can be added.

The MPT taxonomy itself is a useful characterization of systems and software engineering techniques that provides a top-level view of potential MPTs. By viewing the potential MPTs in the taxonomy, the sponsor may choose specific MPTs for further evaluation based on their processes and tools, or the sponsor may eliminate MPTs that involve processes that are obviously impossible to implement in the sponsor environment. However, the taxonomy alone is not sufficient for evaluating candidate MPTs for adoption or for assessing the sponsor's teams' practices. The taxonomy should be a first step when vetting MPT candidates for adoption. The taxonomy depicts critical elements essential to an MPT, but does not describe the interactions between these elements, which are equally critical to successful MPT implementation. The second step to vetting MPT candidates should be a formal processes analysis (e.g. Little-JIL modeling) of the required interactions between the critical elements of the MPT.

The MPT Themes derived as part of taxonomy development will support future efforts in a number of ways. First the MPT themes can be used to support gap analysis in identifying strategies (or elements of these strategies) that currently exist in software engineering domains but which currently have no practical implementation in systems engineering domains. Second, the MPT themes and their elements can be used to guide the development of new MPTs. Finally, multiple themes (i.e. multiple strategies) can be applied to the same challenge area creating a more robust approach to the problem, while identifying MPTs that implement the same theme may be indicators of redundancy and/or waste in the current systems engineering methodology.

The MPT taxonomy would benefit from an interactive representation through a tool. Incorporating the MPTs from the survey responses alone would create an unreadable textual or graphical taxonomy. Also, an interactive tool would enable the presentation of more relationships between the taxonomy elements. For example, different views on the taxonomy could show the input/output relations of Artifacts to Processes, and the responsibility of Agents to perform Processes and process steps. An interactive representation could also be linked to or synchronized with the Little-JIL process modeling to provide the most complete process representation.

While conducting this work, we found many indicators where future work on MPTs will be necessary to truly enable agility, and better facilitate the effective development of systems in tightly constrained, fast-reaction environments. We have compiled a set of necessary elements that must be addressed by any solution intending to provide agility, including themes such as the frequent integration of components and an iterative development cycle that allows for periodic developer feedback. In many cases, these themes act as requirements for MPTs that have not been developed yet.

Our work with the bridge diagrams has demonstrated them to be an effective way of communicating the results of our research, and of understanding effective system engineering MPTs that can fit within the sponsor's context. We have found that these diagrams give us a useful way of understanding those key system engineering themes necessary to address practical problems; once we have a checklist of these themes to look for we can move on to a more detailed discussion of whether the right MPTs have been chosen for effectively addressing those themes.

5.2 Recommendations on Micro-process Modeling

Based upon the results of our research in this area, the following are recommendations regarding subsequent research in this area:

- Elaborate the initial Little-JIL model of the Scrum process to lower levels of detail in order to elucidate more features of this method.

- Define other methods in Little-JIL, especially those suggested for support of agile development, to provide better insight into the nature of these other methods.

- Study the models developed to determine ways in which they might be integrated (as Continuous Integration and Scrum were integrated) to provide better coverage of system development needs and to suggest ways in which gaps might be filled.

- Apply the Fault Tree Analysis and Finite State Verification analysis techniques demonstrated in this report more broadly to existing and proposed new Little-JIL process definitions. These analyses should be aimed at identifying defects and vulnerabilities in these Methods, and at using the identification of these defects to suggest and verify improvements.

- Apply additional analysis techniques such as Failure Mode and Effects Analysis and Discrete Event Simulation to Little-JIL process definitions in order to help determine the effectiveness of these approaches in improving process and method definitions.

- The effectiveness of the various analysis techniques should be compared and contrasted in order to determine the matches for techniques and desired improvements in processes and methods.
- Develop an engine to support the execution of Little-JIL process definitions. This engine will help evaluate and systematically improve systems engineering MPTs as well as support system development. The effectiveness of such automated aids to system development in guiding and assisting humans should then be measured and assessed.

5.3 Recommendations for Future Research

Based on our research and findings, we recommend the following six areas for future research:

1. MPT identification and classification
2. Test bed facility
3. Empirical Studies
4. Innovation teams for new approaches
5. More effective process improvement
6. Characteristics of agile processes

These areas are described more fully in the following sections.

5.3.1 MPT identification and classification

Continue to identify and describe useful MPTs using the expanded taxonomy. As more MPTs are described, the inter-relationships among them become more apparent, and leverage points for targeted research will be identified.

5.3.2 Test bed facility

Investigate the practicality and usefulness of a test bed facility to evaluate incremental improvement of existing MPTs and new approaches to systems engineering, including new MPTs. This would include work with practitioners to develop and test recommendations, both about how existing practices can (possibly with tailoring) help address agility, and to pilot new MPTs so that we can determine whether they are worth further deployment. The research team will carefully vet and evaluate these new ideas, so that the sponsor can avoid spending resources introducing MPTs for which the feasibility and effectiveness are still in question.

The recommendations concerning Little-JIL and automated analysis tools would provide the initial capabilities for the test bed facility. The Air Force Research Laboratory's Spruce project provides some experience and possible additional infrastructure for applying this virtual test bed concept.

5.3.3 Empirical Studies

There is a need to empirically test some of the key questions about MPTs. Two questions in particular that arose in the MPT research provide good examples of this type of inquiry:

1. *Does Scrum scale well to larger teams?* Our work has indicated that the scrum process is one of the most promising mechanisms for helping achieve agility. Its key contribution is the ability to provide rapid feedback to the development team, both in terms of keeping resources effectively focused on the task at hand, and rapidly re-prioritizing tasks as more feedback is received from the customer. As we indicated in the packaging section, however, it is not clear that scrum can easily scale past small team size. The anecdotal evidence is unclear as to whether large teams can be effective with this approach or not, and we could find no quantitative tests. Where successful application of scrum has been described on large teams, it typically relies on a "scrum of scrums" approach. Unfortunately, this MPT has not been well defined and anecdotal evidence can be found to support both its success and failure. Future work could focus on better defining effective MPTs that address the problem of scale, relying on both experiences at proxy organizations and explicit test / observation where appropriate.

2. *How costly is continuous integration?* Continuous integration seems to be another linchpin of agility, especially in software-intensive systems. However, anecdotally we have found many concerns raised about the cost of this practice, and our work so far has not uncovered any good measures or heuristics on that regard. Future work should focus on empirical studies of this process in real development environments to better quantify the effort and costs required to obtain effective results using this practice. The objective would be to determine whether or not the cost is likely to be a prohibitive factor in adopting this MPT, and to give better guidance to teams who do adopt it as to how to schedule for it.

5.3.4 Innovation teams for new approaches

The MPT research identified a number of gaps in the MPT arsenal. Innovation teams, each focused to create, evaluate and if appropriate, pilot new systems engineering approaches and MPTs, could be established to address one or more specific gaps. The SERC represents a large cross-section of the systems engineering research community. The support from the practitioner and innovative community developed in this research provides a foundation for a strong community to participate in the revisioning and transformation of systems engineering.

5.3.5 More effective process improvement

With the recommendations of existing MPTs and the development of new approaches, it is important to revisit the transition, adoption, and continuing evaluation of systems engineering activities in operational environments. Current process improvement paradigms (e.g. CMMI, ISO-9000) have not been shown to meet the needs of current development organizations. This research would investigate new process improvement methods that are applicable to rapid response and innovative environments.

5.3.6 Characteristics of agile processes

To be truly effective, MPTs that support agile systems engineering should be applied in processes that are as agile as the developments they support; that is, the processes need to respond to significant changes in the environment without breaking. Defining characteristics of such agile processes is key to creating, managing, and improving them.

This page intentionally left blank

APPENDIX A - SURVEY FOLLOW-UP INTERVIEW QUESTIONS

SYSTEMS ENGINEERING
Research Center

Interview Sheet

Assessment of Systems Engineering Methods, Processes and Tools (MPTs) to Support Agile System Development

11/16/2009

Background:

This interview is part of a series of research efforts working to address systems engineering shortfalls in projects characterized as quick response, network-enabled, or emergent. You were requested to participant based on your responses to our earlier on-line survey of systems engineering MPTs.

We are looking at a subset of agile tools: scrum, rapid prototyping, and continuous integration. Three types of information are being requested as part of this interview. The first is a description of your product development environment. The second is a description of how you apply these MPTs in practice to help you meet stakeholder, organization and development environment requirements. The third is your perception of the relative strengths and areas for improvement in the MPTs you are using. The questions in the interview are used as a guide only. If you have related comments or suggestions on MPTs please let us know. We will also check to see if you would like a copy of the final report of the interviews.

For more information or to follow up on this work, please contact Rich Turner, rturner@stevens.edu.

Questions:

1. What are the primary stakeholder values (such as schedule, cost, reliability, maintainability, etc.) that drive your product development environment?
2. Please describe the typical projects that you are currently involved with in terms of size, complexity, duration and/or stability of requirements?
   a. Domain – e.g. commercial, defense, contract

   b. Size – e.g. approximate KLOC (thousand lines of code), if applicable
   c. Relative complexity – low, medium, high
   d. Duration – Deliverable cycle and age of product
   e. Requirements stability – stable (up front definition and commitment), infrequent changes, frequent changes
3. How would you characterize your product development environment in terms of information sharing, security and/or interfacing with other systems or system of systems?
4. Does your product involve information sharing with outside systems?
5. Does your product interface/interoperate with other components?
6. Is security a strict requirement for your system? What are the implications of a security breach?
7. Do you use **Scrum** to support the typical projects you are involved in?
   a. How have you had to tailor Scrum for your context?
   b. What challenges have you faced in implementing Scrum in your development environment? Has Scrum influenced your development environment in a positive or negative way?
   c. Has Scrum enabled your team to be successful in meeting the needs of your stakeholders? Why or why not?
8. Do you use **rapid prototyping** to support the typical projects you are involved in?
   a. How have you had to tailor rapid prototyping for your context?
   b. What challenges have you faced in implementing rapid prototyping in your development environment? Has rapid prototyping influenced your development environment in a positive or negative way?
   c. Has rapid prototyping enabled your team to be successful in meeting the needs of your stakeholders? Why or why not?
9. Do you use **continuous integration** to support the typical projects you are involved in?
   a. How have you had to tailor continuous integration for your context?
   b. What challenges have you faced in implementing continuous integration in your development environment? Has continuous integration influenced your development environment in a positive or negative way?
   c. Has continuous integration enabled your team to be successful in meeting the needs of your stakeholders? Why or why not?
10. In your current or expected product development efforts, what challenges exist for which there are no good MPTs?
11. (If time permits.) What other processes, practices, or tools does your team use to manage/deal with changing requirements, stakeholder feedback, integration/interoperability, maintenance?
12. What other comments or suggestions do you have on the MPTs?
13. Would you like a copy of the final report from these interviews?

Email: ___________________________________

APPENDIX B – TAXONOMY CONTENT

TAXONOMY DEFINITIONS

Method (M) – A Method is a collection of inter-related processes, practices, artifacts, agents, resources and tools. A method is essentially a "recipe." It can be thought of as the application of inter-related processes, practices and tools wherein different agents use resources to create and apply artifacts to a class of problems.

Process (P) – A process is a logical sequence of steps (tasks) intended to achieve an objective. The objective achieved may be abstract (e.g. “negotiate among multiple stakeholders”) and/or a composite of multiple individual goals (e.g. “Deliver a fixed-date, variable-scope system”). Performance of a step is often the responsibility of an agent, which may be a human, a device, or a software system. Performing the step may consume resources and require access to various kinds of artifacts in order to execute. Execution of a step will generally produce more artifacts. The structure of a process enables several levels of aggregation (i.e. sub-processes) to allow understanding and analysis of the process at multiple levels of abstraction in support of decision-making.

Tool (T) – A tool automates or partially automates one or more steps within a process and thereby enhances process performance efficiency.

Artifact (Ar) – An artifact is an entity produced by the execution of a process or step. Artifacts are used either as input to another process or step, or as part of the final product produced by the process or method. Artifacts may be tangible objects, but may also be software objects such as files, reports, analyses, or documentation. In the textual taxonomy, related processes that produce and use artifacts are identified.

Resource (R) – A resource is an entity that is required as a support to the performance of a process or step, but which is not a tool or an artifact. For example, a web conferencing tool is a resource that may be used to assist in a planning meeting (but is not considered a Tool in the taxonomy because it does not automate steps within the process of planning). An integration machine that contains a clone of the deployment environment may be a resource used during integration testing. In the textual taxonomy, related processes that use the resources are identified.

Agent (Ag) – An agent is a human actor that is responsible for the performance of a process or step. An agent may require the assistance of resources in order to perform the process or step. Agents are responsible for the completion of the process or step, either by personal activity or by assignment of some or all parts of the step to other agents. In the textual taxonomy, the related processes for which the agents are (partially) responsible for are identified.

SCRUM

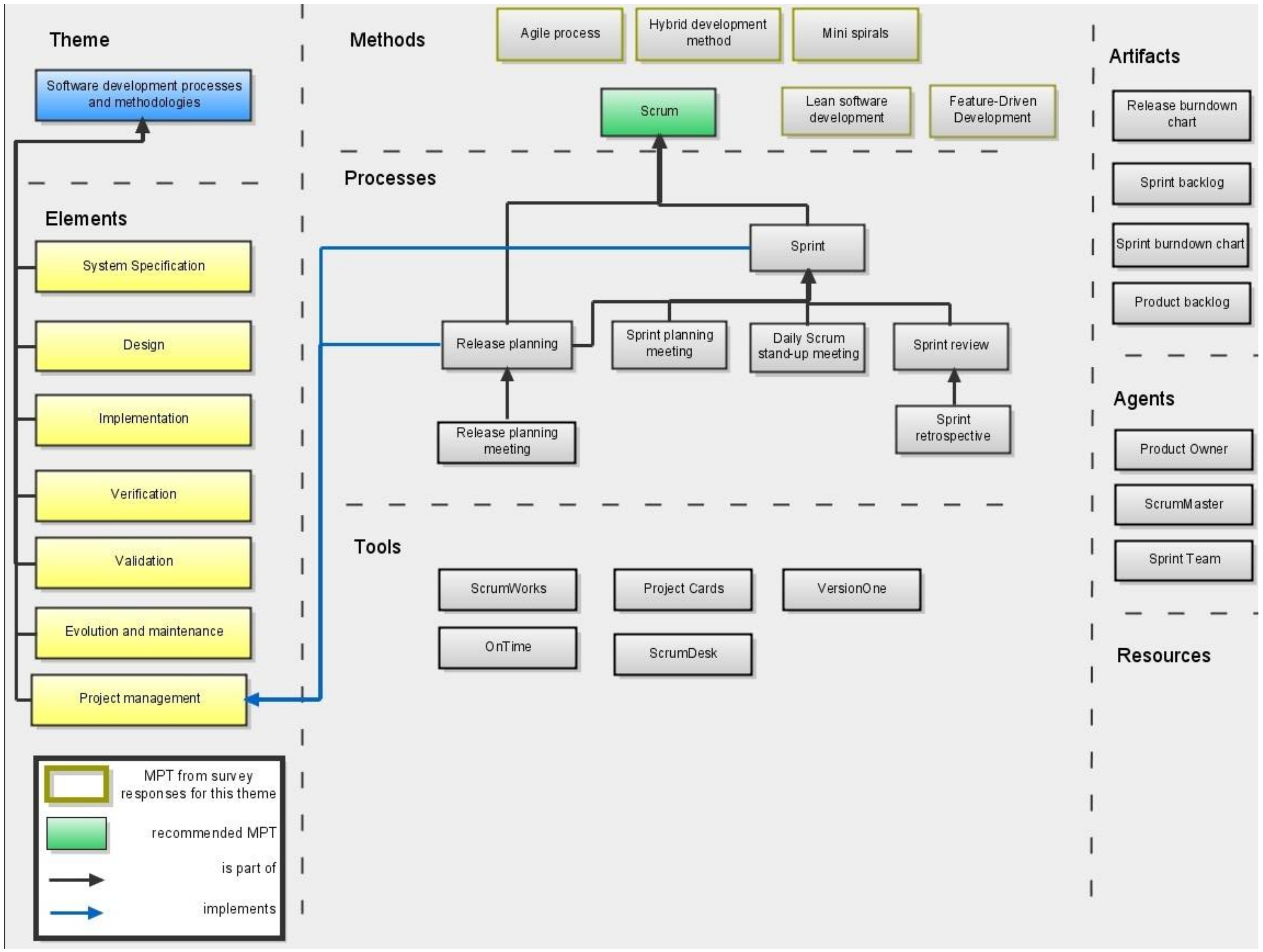

Theme
Software development processes and methodologies
Elements
System Specification
Design
Implementation
Verification
Validation
Evolution and maintenance
Project management
Methods
Agile process
Hybrid development method
Mini spirals
Scrum
Lean software development
Feature-Driven Development
Processes
Sprint
Release planning
Sprint planning meeting
Daily Scrum stand-up meeting
Sprint review
Release planning meeting
Sprint retrospective
Tools
ScrumWorks
Project Cards
VersionOne
OnTime
ScrumDesk
Artifacts
Release burndown chart
Sprint backlog
Sprint burndown chart
Product backlog
Agents
Product Owner
ScrumMaster
Sprint Team
Resources
MPT from survey responses for this theme
recommended MPT
is part of
implements

Figure 4. Scrum Bridge Diagram

Table 4. Scrum Taxonomy

MPT or Theme	*Scrum*
Methods	Scrum
Processes	**Release Planning Meeting** – The purpose of release planning is to establish a plan and goals that the Scrum Teams and the rest of the organizations can understand and communicate; **Sprint** – A Sprint is an iteration. Sprints are time-boxed – A period of 30 days of less where a set of work will be performed to create a deliverable; **Sprint Planning Meeting** – The Sprint Planning meeting is when the iteration is planned. It is time-boxed to eight hours for a one month Sprint. For shorter Sprints, allocate approximately 5% of the total Sprint length to this meeting; **Sprint Review** – This is a four-hour time-boxed meeting for one month Sprints. For Sprints of lesser duration, this meeting must not consume more than 5% of the total Sprint; **Sprint Retrospective** – After the Sprint Review and prior to the next Sprint Planning meeting, the Scrum Team has a Sprint Retrospective meeting. At this three hour, time-boxed meeting the ScrumMaster encourages the Scrum Team to revise, within the Scrum process framework and practices, their development process to make it more effective and enjoyable for the next Sprint; **Daily scrum stand-up meeting** – A daily 15-minute inspect and adapt meeting at which progress and impediments to progress are reviewed); Interviews with practitioners indicated that less frequent meetings (bi-weekly) were associated with difficulty in communicating issues;
Artifacts	**Product backlog** [Produced by: Release Planning Meeting. Used in: Sprint Planning Meeting, Sprint Review] (All work to be performed in the foreseeable future, both well defined and requiring further definition; A prioritized list of everything that might be needed in the product); **Sprint Backlog** [Produced by: Sprint Planning Meeting; Used in: Sprint Review, Sprint Retrospective; Daily stand-up] (Work which is well-enough defined that it can be worked on with relatively little change over a period of 30 days or less and will result in a tangible, incremental deliverable; A list of tasks to turn the Product Backlog for one Sprint into an increment of potentially shippable product); **Burn-Down Charts – [**Produced by: Sprint, Used in: All processes] **Release Burn-Down** measures remaining Product Backlog across the time of a release plan; **Sprint Burndown** measures remaining Sprint Backlog items across the time of a Sprint;
Agents	**Product Owner / Product Manager** [Responsible for steps in: Release Planning Meeting, Sprint Planning Meeting, Sprint] **–** The Product Manager is responsible for maintaining Product Backlog, along with estimates for how much work is required for a backlog item. As the product is built incrementally, the Product Manager re-estimates (sometimes the feature is only partially implemented) or zero's (feature completed)

	backlog items.; **ScrumMaster** [Responsible for steps in: All processes] **–** The ScrumMaster is the person who conducts the Scrum meetings, empirically measures progress, makes decisions, and removes impediments that slow or stop work. This is often the engineering or marketing manager for this product or system area. The Scrum Master is responsible for the daily Scrum meeting. They must ask the following 3 questions at each daily meeting: 1. What have you done since the last scrum meeting? 2. What has impeded your work? 3. What do you plan on doing between now and the next scrum meeting? They are also responsible for making decisions immediately, if required to remove impediments to progress, and noting impediments that must be resolved external to the meeting and causing them to be removed. Once the sprint is underway, new backlog cannot be added to the Sprint except when the ScrumMaster determines that a new backlog item will enhance the viability of the product, is in alignment with the sprint, builds on the sprint's executable, and can be completed within the sprint's time frame. **Sprint Team (Cross Functional)** [Responsible for steps in: All processes] **–** The sprint team has final say in estimating and determining what they can accomplish during the sprint. A useful link that discusses the 3 different roles: http://www.scrumalliance.org/pages/scrum_roles
Resources	
Tools	Danube – ScrumWorks Pro / ScrumWorks Basic; Axosoft – OnTime; VersionOne; AgileBuddy; Acunote; BananaScrum; FireScrum; ThoughtWorks – Mingle; Project Cards; Scrummy Pro; ScrumDesk; Inflectra – SpiraPlan; Scrum For Team System – TaskBoard; TinyPM; Protonotes

CONTINUOUS INTEGRATION

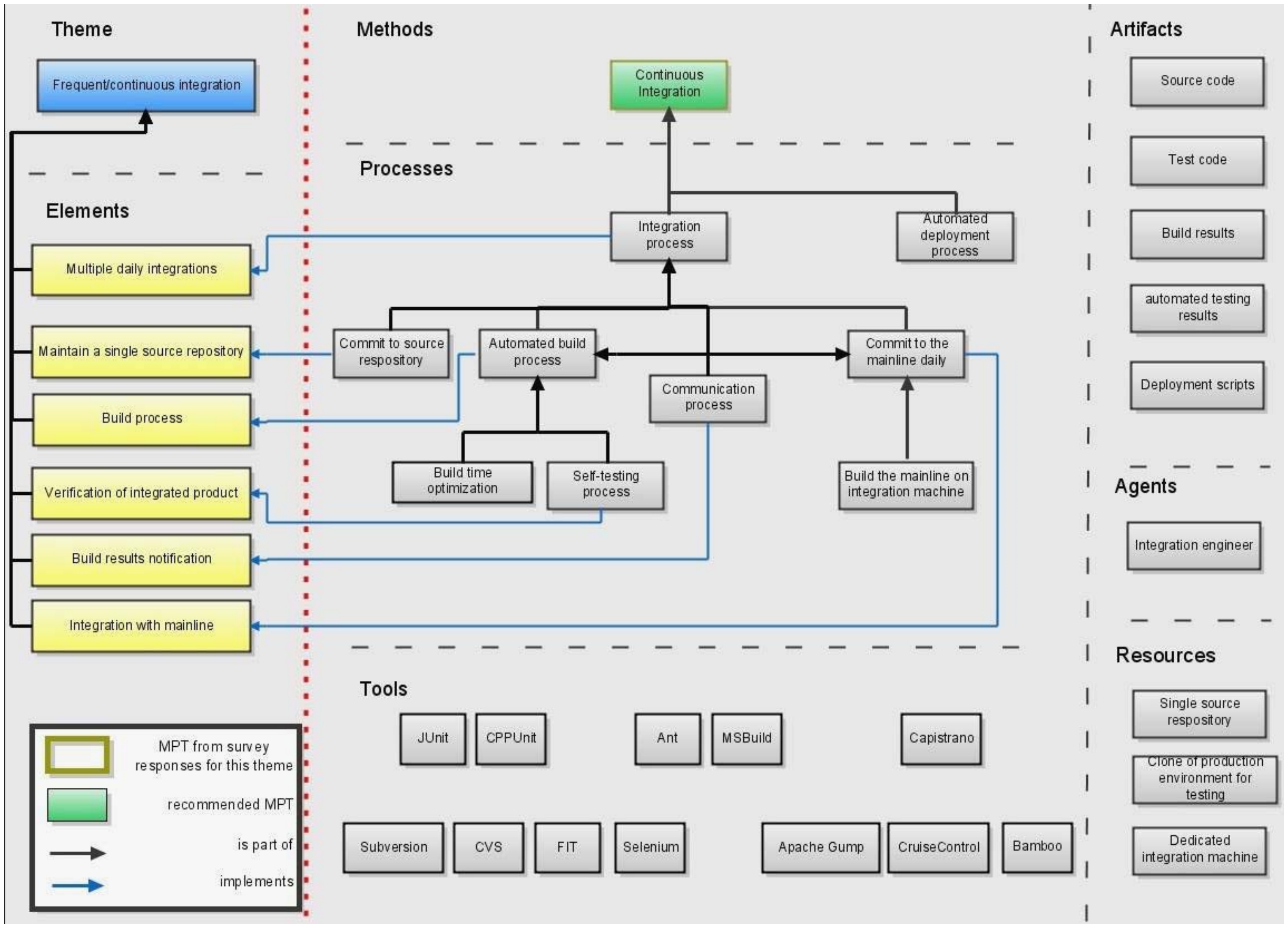

Theme
Frequent/continuous integration
Elements
Multiple daily integrations
Maintain a single source repository
Build process
Verification of integrated product
Build results notification
Integration with mainline
MPT from survey responses for this theme
recommended MPT
is part of
implements
Methods
Continuous Integration
Processes
Integration process
Automated deployment process
Commit to source respository
Automated build process
Communication process
Commit to the mainline daily
Build time optimization
Self-testing process
Build the mainline on integration machine
Tools
JUnit
CPPUnit
Ant
MSBuild
Capistrano
Subversion
CVS
FIT
Selenium
Apache Gump
CruiseControl
Bamboo
Artifacts
Source code
Test code
Build results
automated testing results
Deployment scripts
Agents
Integration engineer
Resources
Single source respository
Clone of production environment for testing
Dedicated integration machine

Figure 5. Continuous Integration Bridge Diagram

Table 5. Continuous Integration Taxonomy

MPT or Theme	*Continuous Integration*
Methods	Continuous integration
Processes	**Integration process** – Developers integrate their code multiple times daily with a copy from the mainline, verifying their code against the build tests **Commit to source repository –** Developers commit their code to a single source version control system, e.g. Subversion, CVS, Software Configuration and Library Manager (IBM) **Automated build** – The system is built automatically when code is committed by developers **Self-testing build** – Upon system build, the integration machine executes a test suite that performs automated verification (unit, integration and systems tests); **Commit to the mainline daily** – After developers have tested their changes against the mainline, changes are committed to the mainline so that changes are communicated to other developers daily (capture changes made by other developers in the time you have been working on your version); **Build the mainline on an integration machine** – Once the developer commits, a final integration build against the mainline (which contains the latest commit) is triggered on an integration machine. This final build ensures there are no conflicts with changes from another developer and to eliminate environmental difference between the developers' machines. **Build time optimization** – use engineering practices to keep the build time low to provide rapid feedback to developers on broken builds or failed tests (e.g. using a staged build) **Communication process** – ensure that everyone can easily see the state of the mainline and the changes that have been made to it. Manual builds should have results communicated to developers, while automated builds may send communications or provide a dashboard of build results. **Automated deployment process** – have scripts that will allow you to deploy the application into any environment easily; also consider rollback.
Artifacts	**Source code** [Produced by: Commit to the source repository; Used in: All processes] **Test code** [Produced by: Commit to the source repository; Used in: Automated build, Self-testing build, Commit to the mainline, Build the Mainline on an Integration Machine]; **Build results** [Produced by: Automated build; Used in: Communication process]; **Automated testing results** [Produced by: Self-testing build; Used in: Communication process]; **Deployment scripts** [Used in: Automated build; Build the mainline on an integration machine; Automated deployment process]
Agents	Integration engineer [Responsible for steps in: Automated build; Self-testing build; Build the mainline on an integration machine; Build time optimization; Communication process; Automated deployment process]
Resources	**Clone of the production environment for testing** [Used in: Automated build; Self-testing build; Build the mainline on an integration machine]– Integration testing and builds should be performed in a clone of the production environment

	dedicated integration build machine
Tools	**Continuous Integration** – AccuRev, Anthill Pro, Apache Continuum, Apache Gump, Automated Build Studio, Bamboo, CABIE, Cascade, Cerberus, ControlTier, CruiseControl, Cruise, CruiseControl.NET, CruiseControl.rb, Draco.NET, FinalBuilder, Hudson, Parabuild, TeamCity, Team Foundation Server **Automated Build** – Java (Ant, Ruby, etc), .NET (Nant, MSBuild, etc) **Self Testing** – XUnit (Java – JUnit), FIT, Selenium, Sahi, Watir, FITnesse **Automate Deployment (and Rollback)** – Capistrano

RAPID PROTOTYPING

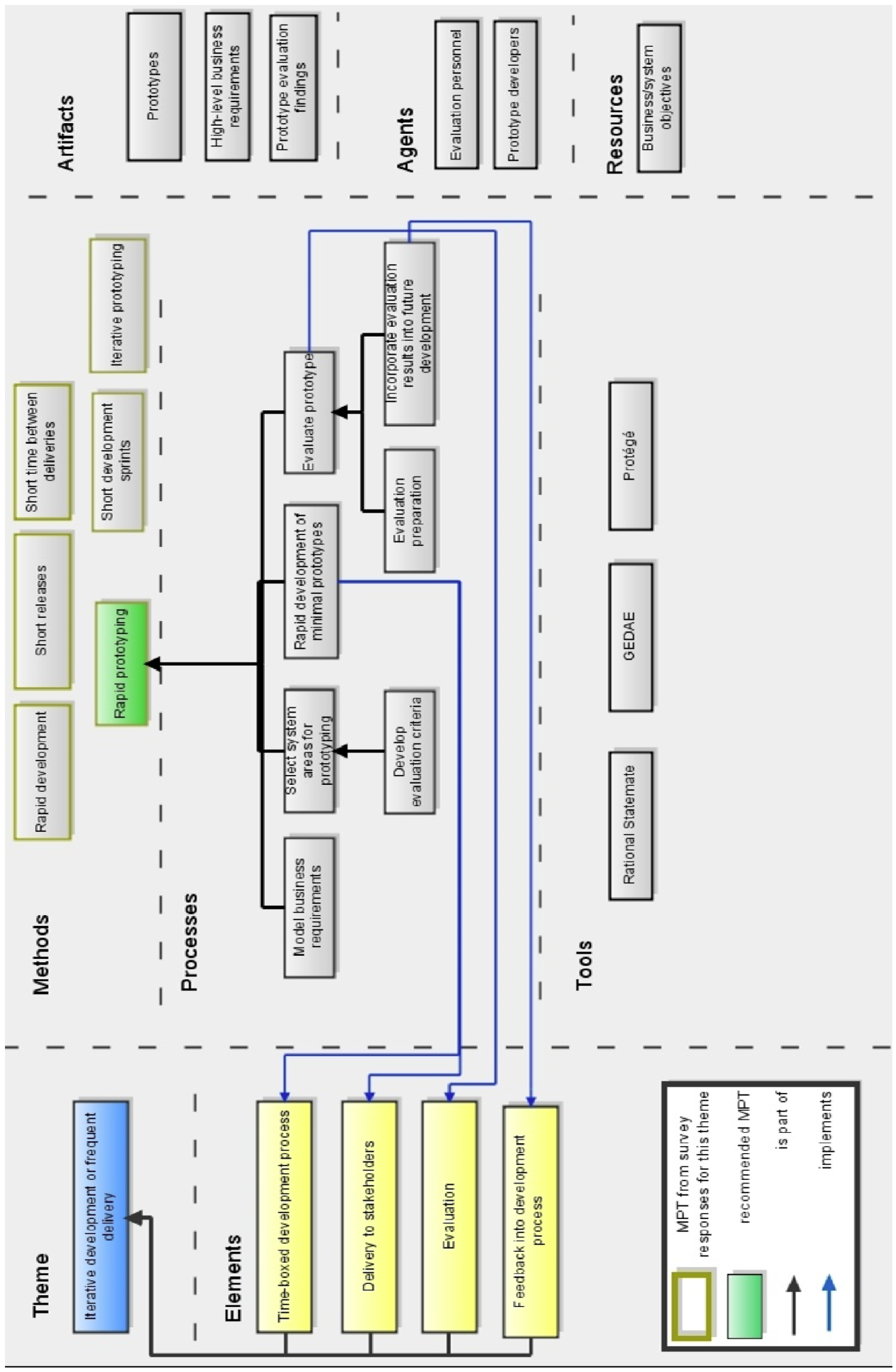

Theme
Iterative development or frequent delivery
Elements
Time-boxed development process
Delivery to stakeholders
Evaluation
Feedback into development process
Methods
Rapid development
Short releases
Short time between deliveries
Rapid prototyping
Short development sprints
Iterative prototyping
Processes
Model business requirements
Select system areas for prototyping
Rapid development of minimal prototypes
Evaluate prototype
Develop evaluation criteria
Evaluation preparation
Incorporate evaluation results into future development
Tools
Rational Statemate
GEDAE
Protégé
Artifacts
Prototypes
High-level business requirements
Prototype evaluation findings
Agents
Evaluation personnel
Prototype developers
Resources
Business/system objectives
MPT from survey responses for this theme
recommended MPT
is part of
implements

Figure 6. Rapid Prototyping Bridge Diagram

Table 6. Rapid Prototyping Taxonomy

MPT or Theme	*Iterative development or frequent delivery*
Methods	Rapid Prototyping
Processes	**Model business requirements** – Create a specification (likely informal) of the business objectives for the system. These business objectives are used to generate functional scenarios, key data elements and interactions with other systems. **Select system areas for prototyping** – Select the system areas that will be prototyped to inform decision makers. Potential areas for prototyping include: desired functionality, technology selection (e.g. which data base vendor to use), and user interfaces. **Develop evaluation criteria** – Determine and document the criteria against which the prototypes will be evaluated. **Rapid development of minimal prototypes** – Rapidly create prototypes that can be evaluated by decision makers. Prototypes may be created using presentation slides, visual drawings, mock user interfaces in a programming language, etc. Prototypes should provide enough information for decision makers to achieve their goal of evaluating the desired functionality, technology selection, user interface, or other system area. **Evaluate prototype** – Decision makers evaluate the prototype against a set of criteria. Evaluation may occur through presentation, analysis, and/or use. **Evaluation preparation** - Prepare necessary artifacts for evaluation (e.g. tasks that users will perform while using the prototypes) and determine/solicit participants where necessary. **Incorporate evaluation results into future development** – The results of the evaluation are captured and used to inform further prototyping or to guide the development of more complete system specification.
Artifacts	**Prototypes** [Produced by: Rapid development of minimal prototypes; Used in: Evaluate prototype; Evaluation preparation] **High-level business requirements** [Produced by: Model business requirements; Used in: Select system areas for prototyping, Develop evaluation criteria; Rapid development of minimal prototypes] **Prototype evaluation findings** [Produced by: Evaluate prototype; Used in: Incorporate evaluation results into future development]
Agents	**Evaluation personnel** [Responsible for steps in: Evaluate prototype]; **Prototype developers** [Responsible for steps in: Rapid development of minimal prototypes; Evaluation preparation]
Resources	**Business/system objectives** [Used in: Modeling business requirements; Select system areas for prototyping; Develop evaluation criteria]
Tools	**Prototype modeling, development and analysis -** Rational Statemate - (a graphical design, simulation, and prototyping tool for the rapid development of complex embedded systems); GEDAE (Advanced DSP software development tool. It enables rapid prototyping, speeds development and analysis, simplifies optimizations, and increases performance and flexibility); Protégé (Protégé is a free, open source ontology editor and knowledge-base framework. It provides a plug-and-play environment that makes it a flexible base for rapid prototyping and application development.)

APPENDIX C – LITTLE-JIL MODELS

DEFINING PROCESSES WITH THE LITTLE-JIL PROCESS DEFINITION LANGUAGE

Little-JIL is a process definition language [Wise 2006] that, along with its interpreter Juliette [Cass 2000], supports specification and execution of processes involving different agent and non-agent resources. A Little-JIL process definition is comprised of four orthogonal components: 1) a coordination specification, 2) a resource specification that includes constraints, 3) a specification of artifacts and their flow and 4) a specification of the behaviors of those resources that can be assigned as agents.

The most immediately noticeable aspect of a Little-JIL process definition is the visual depiction of the coordination specification. This components of the Little-JIL process definition looks initially somewhat like a task decomposition graph, in which processes are decomposed hierarchically into steps. The steps are connected to each other with edges that represent both hierarchical decomposition and artifact flow. Each step contains a specification of the type of agent resource needed in order to perform the task associated with that step. Thus, for example, in the context of a software development process, the agents would be entities such as programmers, testers, managers, the customer, etc. The collection of steps assigned to an agent resource defines the interface that the agent must satisfy to participate in the process. It is important to note that the coordination specification includes a description of the external view and observable behavior of such agent resources. But a specification of how the agent resources themselves perform their tasks (their internal behaviors) is not a part of the coordination specification, but rather is the fourth components of a Little-JIL process definition. It is important to note that Little-JIL enforces a sharp separation of concerns, separating the internal specification of what a resource is capable of doing and how the agent will do it, from the specification of how agents are to coordinate their work with each other in the context of carrying out the overall process.

The central construct of a Little-JIL process definition is a step. Steps are organized into a hierarchical tree-like structure. The leaves of the tree represent the smallest specified units of work, each of which is assigned to an agent resource that has characteristics consistent with those defined as part of the definition of the step. The tree structure defines how the work of these agent resources will be coordinated. In particular, the agent assigned responsibility for executing a parent node is responsible for coordinating the activities of the agents assigned to execute all of the parent's children. Figure 7 shows the graphical representation of a Little-JIL step with its different badges and possible connections to other steps.

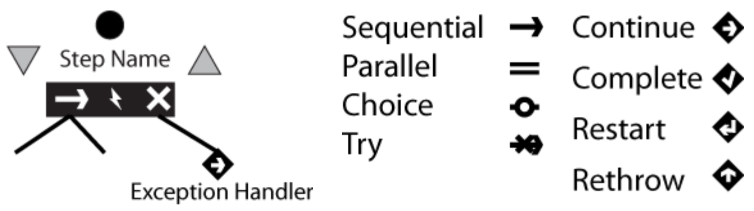


Figure 7. Little-JIL iconography

The *interface badge* is a circle on the top of the step name that connects a step to its parent. The interface badge represents the specification of any and all artifacts that are either required for, or generated by, the step's execution. The interface badge also represents the specification of any and all resources needed in order to support the execution of the step. Chief among these resources is the single resource designated as the step's agent. Below the circle is the step name. A step may also include pre-requisite and/or post-requisite badges, which are representations of steps that need to be executed before and/or after (respectively) this step for the proper performance of the step's execution. A simple form of pre and post-requisites can be simple predicates that need to be evaluated by the process execution engine. A pre-requisite is shown with an upside down triangle on the left of a step bar. Similarly a post-requisite is shown with a regular triangle on the right of a step bar. Inside the central black box of the step structure, there are three more badges. On the left is the control flow badge, which specifies the order in which the child sub-steps of this step are to be executed. A child (substep) of a step is connected to the parent by an edge emanating from the parent and terminating at the child. Artifact flows between the parent and child are indicated by annotations on this edge.

On the right of the step bar is an X, which represents the exception handler capabilities of the step. Below this badge are exception edges that are connected to any and all handlers defined to deal with exceptions that may occur in any of the descendants of this step. Each handler itself a step (that may sometimes be defined to be null), and is annotated to indicate the type of exception that it handles. Here too, artifact flow between the parent and the exception handler step is represented by annotations on the edge connecting them. This edge also bears an annotation indicating the type of exception handled. In the middle of the step bar is a lightning bolt icon, which represents the message handling capabilities of the step. Attached to this badge by message handling edges (also known as reaction handling edges) are any and all handlers defined to deal with messages that may emanate from any step in the process definition. A message can be generated from outside the process as well. The message handling capability is quite similar to the exception handling capability, but while exception handlers respond only to exceptions thrown from within their substep structure (a scoped capability), message handlers can respond to messages generated anywhere (an unscoped capability). If there are no child steps, message handlers, or exception handlers, the corresponding badges are not depicted in the step bar.

One of the important features of the language is its ability to define control flow. There are four different non-leaf step kinds, namely *sequential*, *parallel*, *try*, and *choice*. Children of a sequential step are executed one after another from left to right. Children of a parallel step can be executed in any order, including in parallel, depending on when the agents actually pick up, and begin execution of, the work assigned in those steps. A try step attempts to execute its children one by one starting from the leftmost one and considers itself completed as soon as one of the children successfully completes. Finally a choice step allows only one of its children to execute, with the choice of which child being made by the agent assigned to execute the step.

The pre-requisites and post-requisites associated with each step act essentially as guards, defining conditions that need to hold true for a step to begin execution or to complete successfully. Exceptions and handlers are control flow constructs that augment the step kinds. The exceptions and exceptions handlers work in a manner that is similar in principle to the way in which they work in well-known contemporary application programming languages such as Java. Exceptions indicate an exceptional condition or error in the process execution flow, and handlers are used to recover from, or fix, the consequences of those situations. When an exception is thrown by a step, it is passed up the tree hierarchy until a matching handler is found. Handler steps are annotated with control flow semantics that indicate how program control flow will continue once a raised exception has been handled by the defined handler. Figure 7 shows four different types of continuation semantics for handlers. With these semantics, a process definer can specify whether a step will continue execution, successfully complete, restart execution at the beginning, or rethrow the exception for a higher level parent step to handle.

As noted above, a Little-JIL process definition represents a process as a hierarchical decomposition, where each step can be viewed as a procedure. Thus a key part of a Little-JIL step definition is the representation of the artifacts that function as arguments to the step. This argument specification is represented iconically as part of the interface badge located atop the step's iconic representation. Typical step artifacts are entities such as data items, files, or access mechanisms. Artifacts are passed between steps by two different mechanisms. Most commonly artifacts are passed between parent and child steps very much the way arguments are passed from a calling procedure to a called procedure. This hierarchical argument passing mechanism is complemented by Little-JIL's channel construct, a non-hierarchical argument passing mechanism. Channels can be defined between any set of steps that provide argument artifacts to any set of steps that are consumers of these artifacts. In the examples that follow channels are generally defined between one source step and one destination step. Channels are useful both for the actual communication of artifacts between steps, but also as conduits of signals that can be used for the synchronization of steps that execute in parallel with each other.

Also represented as part of the external interface icon are the resources that are required by a step in order to support its execution. Every step has at least one resource that is required, namely the agent, described above. But some steps may also require additional resources, such as databases, tools, and people who are needed in order to assure that step execution proceeds as it should. All of these resources as specified as types, and a resource management capability to be used in conjunction with Little-JIL process execution is expected to accept requests for resources as type specifications, and then return resources instances that are of the requested type. Clearly this requires the definition of a resource request language that is common to both the Little-JIL process definition and to the resource management facility that is used to support process execution.

Juliette is the execution framework that is used to execute processes written in Little-JIL. Juliette executes Little-JIL process definitions by interpreting steps according to specified sequences. While interpreting a step, Juliette makes requests to an accompanying resource manager to obtain the resources that are required for the execution of that step. The Juliette interpreter then notifies the selected agent by putting the tasks to be done on the agent's to-do-list (agenda). This is done using a distributed Agenda Management System [Mccall 1998]. An agent, in turn, decides which task to pick up from the list of tasks waiting on its agenda. It is expected that agents know how to perform any task that can appear on the agent's agenda, and that the agent will notify the interpreter when a selected task has been completed. The resource (and thus agent) definition, as mentioned above, is separate from, and orthogonal to, the Little-JIL coordination definition. How an agent carries out a particular task is independent of the coordination dictated by the process.

We present some definitions of agile system development methods that have been written using the Little-JIL process definition language. Additional features of Little-JIL will be described in the context of explaining some of the key details of these models.

We begin by presenting a Little-JIL definition of the Scrum software development method. As suggested by the preceding discussion, this process will be defined hierarchically. We begin with the top-level definition of the Scrum process.

The Little-JIL Scrum definition represents Scrum as a decomposition into three substeps executing in parallel: **Manage Product Backlog**, **Manage Product Release**, and **Manage Product Development. Manage Product Development** is, in turn, decomposed further as a series of different instances of the **Development Iteration** step. The fact that **Manage Product Development** is decomposed in this way is indicated as follows. The attachment of + sign to the edge connecting **Manage Product Development** to **Development Iteration** indicates that one or more instances of **Development Iteration** are the children of **Manage Product Development.** And the right arrow icon in the **Manage Product Development** step indicates that these substeps are to be executed sequentially.

SCRUM IN LITTLE-JIL

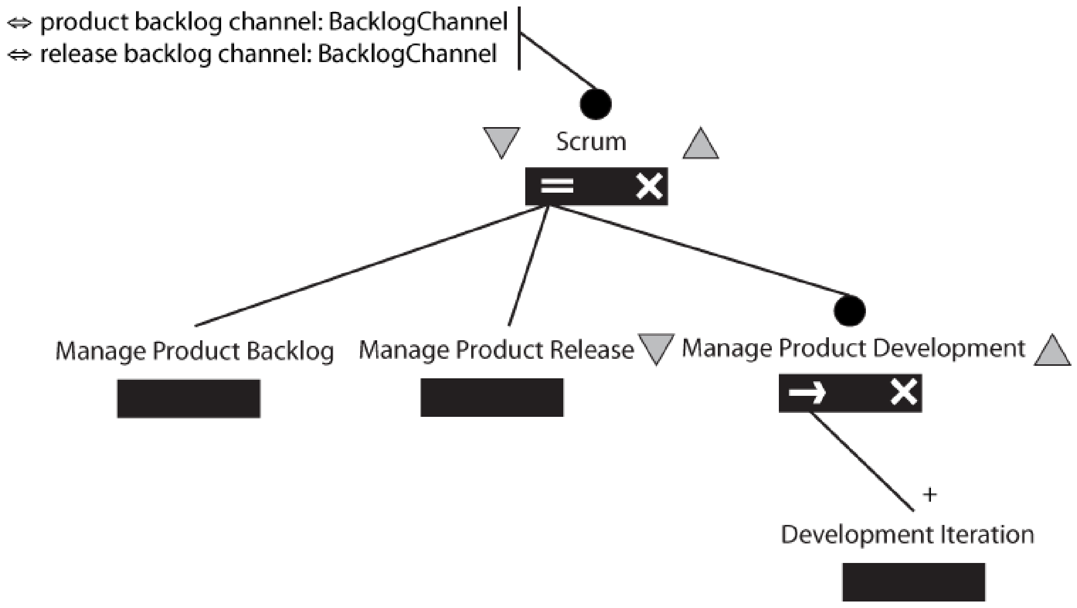

⇔ product backlog channel: BacklogChannel
⇔ release backlog channel: BacklogChannel
Scrum
Manage Product Backlog
Manage Product Release
Manage Product Development
+
Development Iteration

Figure 8. Top level of the Scrum process

These three top-level Scrum sub-processes communicate with each other via two channels, the product backlog channel and the release backlog channel (identified as channels by means of the double-headed horizontal arrows in Figure 8) that hold product and release backlog information. Each of these sub-processes, and the ways in which they use these channels to communicate with each other are described in more detail below.

Manage Product Backlog

The **Manage Product Backlog** step (Figure 9) is carried out as a sequence of instances of the **Update Product Backlog** step, each of which performs some kind of modification to the *product backlog* artifact, which is the comprehensive list of the work to be done on the system being developed. The agent that carries out the **Manage Product Backlog** step and all instances of the **Update Product Backlog** step is the *product owner* (designated by the annotation “agent: ProductOwner” attached to the dot atop the **Manage Product Backlog** step).

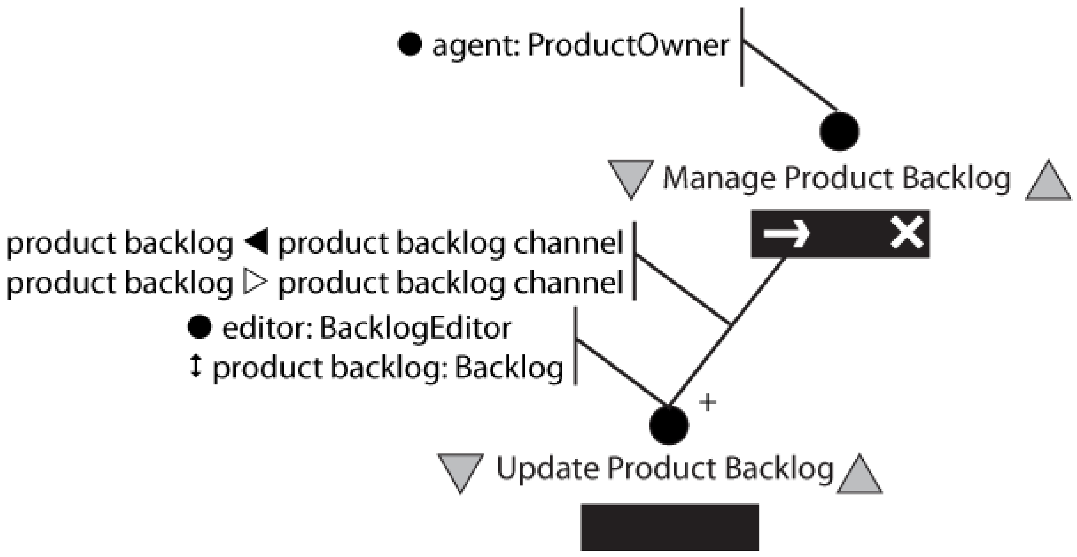

● agent: ProductOwner
Manage Product Backlog
product backlog ◀ product backlog channel
product backlog ▷ product backlog channel
● editor: BacklogEditor
↕ product backlog: Backlog
+
Update Product Backlog

Figure 9. Elaboration of the Manage Product Backlog step

The *product backlog* artifact is taken from the product backlog channel and passed to the **Update Product Backlog** step for processing, and written back after processing has been done by virtue of its being removed (indicated by the solid left-facing arrowhead annotation) from the *product backlog* channel by the **Update Product Backlog** step. Moreover, the fact that the updated product backlog artifact is returned to the product backlog channel is indicated by the right facing unfilled arrowhead.

Manage Product Release

The **Manage Product Release** step (Figure 10) is responsible for specifying how the product backlog is to be partitioned into product releases. This is done as the concurrent execution by the *product owner* of multiple instances of two substeps, **Update Release Backlog** and **Release Planning Meeting.** In the **Manage Release Backlog** step, the *product owner* selects from the *product backlog* the items that are to be included in the next product release by placing those items in the *release backlog*. The + sign on the edge connecting this step to its parent indicates that this step is done one or more times.

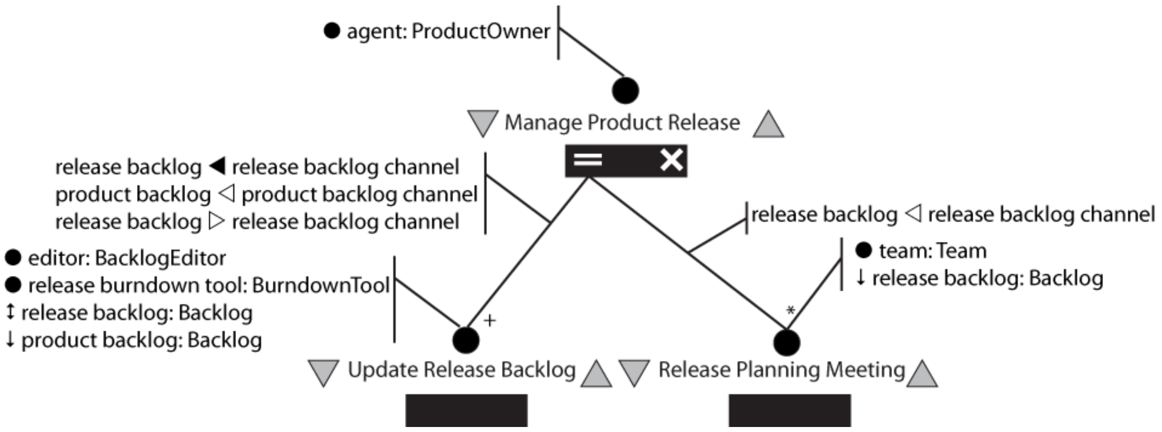

agent: ProductOwner
Manage Product Release
release backlog ◀ release backlog channel
product backlog ◁ product backlog channel
release backlog ▷ release backlog channel
editor: BacklogEditor
release burndown tool: BurndownTool
↕ release backlog: Backlog
↓ product backlog: Backlog
release backlog ◁ release backlog channel
team: Team
↓ release backlog: Backlog
+
*
Update Release Backlog
Release Planning Meeting

Figure 10. Elaboration of the Manage Product Release step

As was the case with the management of the *product backlog* in the **Manage Product Backlog** step, a channel, the *release backlog channel*, is used to manage the *release backlog*, with artifacts being removed from the channel for consideration and processing (filled left arrowhead), and then passed as arguments to the **Update Release Backlog** step. After the step is completed, these artifacts are passed back out of the **Update Release Backlog** step and replaced in the channel (unfilled right arrowhead). Because changes to the release backlog do not modify the product backlog, the product backlog is accessed using a channel read (unfilled left arrowhead), and the artifacts so accessed as passed into the **Manage Release Backlog** step through an input-only argument. This step also requires the use of a *release burndown* tool, which is indicated as a needed resource used to track the amount of work planned for the release against the time left before the release date.

The **Release Planning Meeting** step is carried out zero or more times (note the * on the edge connecting it to its parent). This step represents carrying out a meeting between the *product owner* and the team to ensure that all hands know the release schedule. Note that the *release backlog* artifact is passed into this step as an argument in order to make the release backlog items available for this meeting. Scrum does not specify the frequency or duration of release planning meetings.

Development Iteration

The heart of the Scrum process is the iterative performance of sprints. The **Development Iteration** step (Figure 11) specifies how one of these iterations is to be carried out. An iteration begins with a **Sprint Planning Meeting** to determine the work to be performed during the current iteration. This body of work is represented by the *sprint backlog* artifact. The **Sprint Planning Meeting** step is followed by the **Sprint** step in which the work is actually performed. The iteration concludes with the **Sprint Review** step and the **Sprint Retrospective** step. The **Sprint Planning Meeting** step and the **Sprint** step are elaborated below.

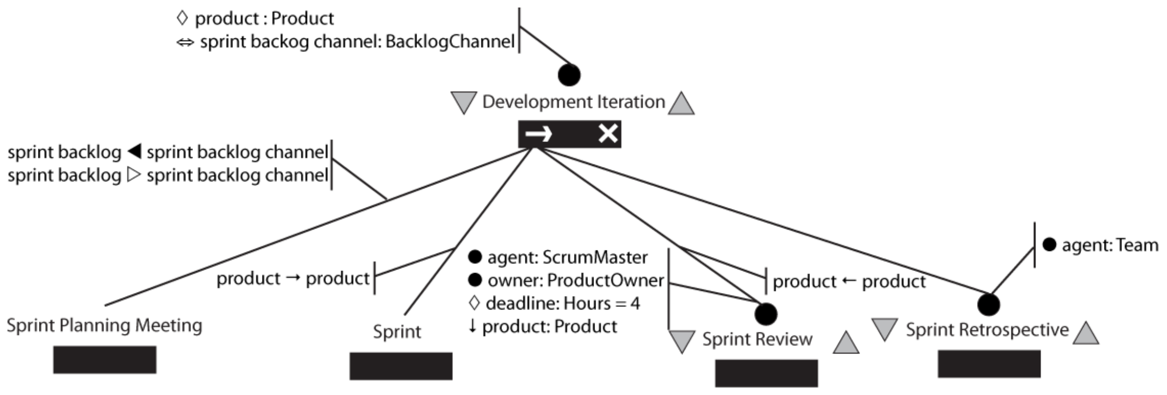


Figure 11. Elaboration of the Development Iteration step.

The **Sprint Review** step is a time-boxed meeting (note the diamond annotation that specifies a fixed deadline for completion of this step) led by the *ScrumMaster* agent with the support of the *ProductOwner* and team as *resources*. This step takes as an input artifact (note downward arrow annotation) the *product* artifact that was produced as the output of the **Sprint** step. The purpose of this **Sprint Review** step is to enable the team to discuss the results of the preceding sprint, and to close the loop between the *product owner* and the *team*. After the **Sprint Review** step concludes, the **Sprint Retrospective** is carried out with the *team* as the agent, indicating that the team meets in private to assess its performance during the last sprint as preparation for the next sprint.

Sprint Planning Meeting

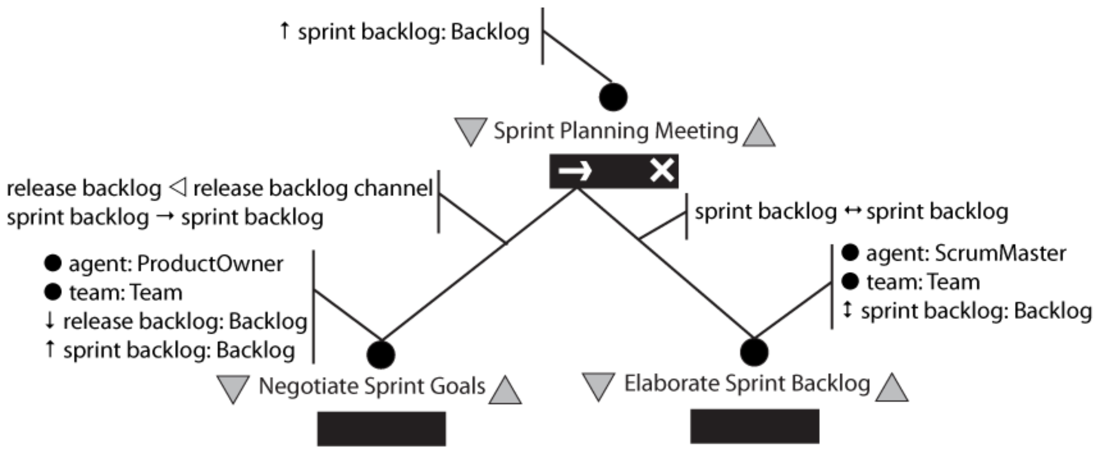

↑ sprint backlog: Backlog
Sprint Planning Meeting
release backlog ◁ release backlog channel
sprint backlog → sprint backlog
sprint backlog ↔ sprint backlog
● agent: ProductOwner
● team: Team
↓ release backlog: Backlog
↑ sprint backlog: Backlog
● agent: ScrumMaster
● team: Team
↕ sprint backlog: Backlog
Negotiate Sprint Goals
Elaborate Sprint Backlog

Figure 12. Elaboration of the Sprint Planning Meeting step

The **Sprint Planning Meeting** step (Figure 12) is elaborated here as a second-level decomposition of the **Manage Product Development** step, but its execution immediately follows the **Manage Product Release** step (see Figure 8). The purpose of this step is to define and elaborate the *sprint backlog* artifact, which is a specification of what the team is committing to complete by the end of the upcoming sprint (note that the sprint backlog artifact is show as an output artifact [upward arrow] from the **Sprint Planning Meeting** step). This step consists of two substeps executed in sequence. The first step, **Negotiate Sprint Goals**, is led by the *ProductOwner* agent, supported as a resource by the *team*. In this step the *release backlog* artifact is retrieved from the *release backlog channel* (left-facing hollow arrowhead) and passed as an input argument. During this step the *Product Owner* identifies the highest priority items in the *release* backlog and negotiates with the *team* to determine which items are to be done.

In the second part of the meeting, represented by the **Elaborate Sprint Backlog** step, the *ScrumMaster* agent leads a discussion, supported by the *team* as a resource, that results in decomposing the items comprising the *sprint backlog* artifact into tasks that are expected to take no more than 1 day to complete. The resulting artifact is then passed out of the **Elaborate Sprint Backlog** step, and then out of its parent, the **Sprint Planning Meeting** step. It will then become an input artifact to the **Sprint** step, to be described next.

Sprint

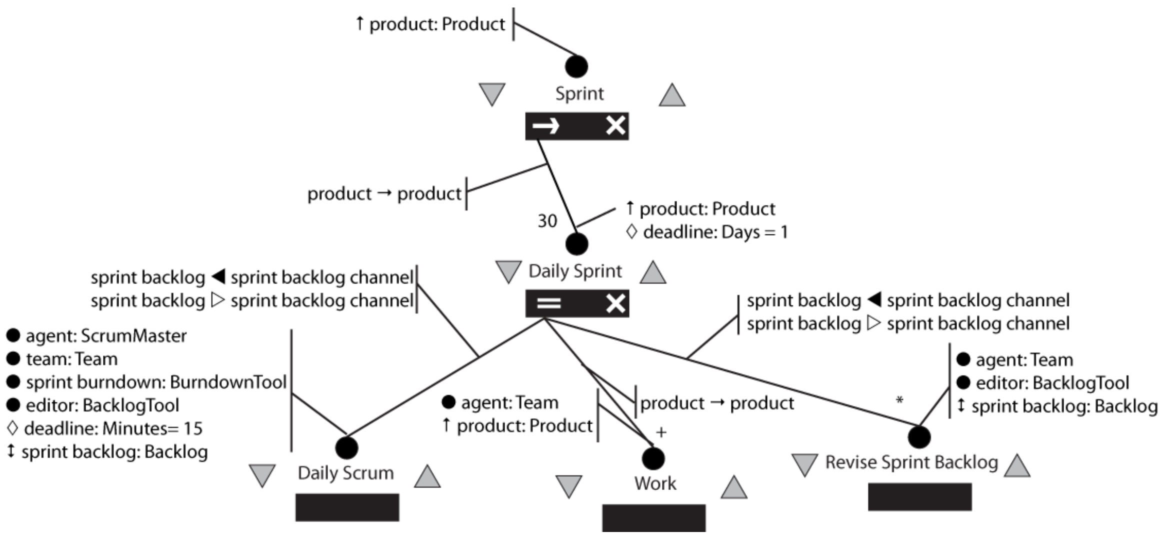


Figure 13. Elaboration of the Sprint step

The **Sprint** subprocess (Figure 13) is the heart of the Scrum process, being the activity during which actual development work gets done. To be more precise, the **Sprint** process consists of 30 (note the 30 annotation on the edge connecting the **Sprint** parent step to its **Daily Sprint** child step) consecutive (note the right arrow step kind badge in the **Sprint** step) performances of the **Daily Sprint** subprocess. As indicated by the = sign badge in the **Daily Sprint** step, this subprocess is carried out as the parallel performance of its three substeps, **Daily Scrum, Work,** and **Revise Sprint Backlog.** Both the **Daily Scrum** and the **Revise Sprint Backlog** steps require both access to and update capability for the *sprint backlog*, with the **Daily Scrum** step consulting the *sprint backlog* to support determination of specific work to be done, and the **Revise Sprint Backlog** step requiring access in order to update the *sprint backlog* to reflect work that has been completed. These accesses for these two steps are coordinated by using the *sprint backlog channel* to provide the needed concurrent access permissions.

The **Daily Scrum** step is a 15 minute (note the specification of this deadline by means of the diamond annotation) progress meeting during which the team meets to identify the product that is to be developed. Note that the *sprint backlog* artifact is passed in as a parameter, and then also passed out of this step, after which it is written to the *sprint backlog channel* so that it is made available to the **Revise Sprint Backlog** step, which may be executing in parallel.

After execution of the **Daily Scrum** step, there are multiple performances of the **Work** step (note the + sign on the edge connecting the **Work** step to its parent). Each instance of the **Work** step takes the *product* artifact as an argument and passes it back out, presumably with the *product* artifact being comprised of more completed work items after the execution of this step. The agent for this step is the *team*.

Concurrently with the performances of the **Work** step there are multiple performances of the **Revise Sprint Backlog** step (note the * on the edge connecting this step to its parent). The agent for this step is also *team*, and the effect of a performance of this step is to update the *sprint backlog* to reflect the completion of work items enumerated there. There is no fixed requirement for the frequency or periodicity of the **Revise Sprint Backlog** step.

Summary: The foregoing Scrum method definition in Little-JIL has illustrated the way in which a language such as this can clarify the relations among the processes, subprocesses, tools, agents, artifacts, and resources that comprise the method. With this particular definition formalism it is easy to see such relations as the way that various processes comprise the method, the ways that subprocesses comprise the processes, the ways that the processes and subprocesses generate and consume artifacts and products, the identities of the various agents and their responsibilities for performing specific processes, subprocesses and steps, and the needs for mechanisms to coordinate concurrent access to key artifacts.

It is important to emphasize that a clear graphical notation such as the one shown here can do much to support understandings of these relations. But it also is important to note that the depiction is only a projection of an underlying method definition that is specified using a rigorously defined process language. The existence of such an underlying rigorous language means that disputes, disagreements, or misunderstandings of the graphical notation can be resolved by consultation with, and reasoning about, the underlying rigorous definition, with the semantics of that definition being used as the basis for needed reasoning. The Little-JIL process definition language, for example, is rigorously defined through the use of finite state machines that provide the precise semantic meanings of each of the language's different step kinds.

In the next section we will see that rigor in such a process definition language can also be useful in supporting powerful analyses of processes written in that language.

ANALYZING PROCESSES DEFINED WITH A MICRO-PROCESS DEFINITION LANGUAGE.

The previous section has demonstrated the precision that can be obtained by using a sufficiently powerful process definition language to capture details of a system development method such as the Scrum. But, as noted above, the use of such a language to define such processes is of additional value in serving as the basis for powerful analyses that can lead to greater insights into such processes, and for the detection of defects and vulnerabilities in such processes.

In this section we illustrate this by demonstrating how the Little-JIL Scrum method definition can be used as the basis for analyses that verify desirable properties, support integration with other methods and indicate vulnerabilities.

Using Fault Tree Analysis to identify and remove vulnerabilities.

Fault tree analysis (FTA) is an analytic approach that is well known in many traditional engineering disciplines, where it is used to identify the ways in which a specified hazard might arise during the performance of a process [Chen 2006]. More specifically, in this approach a graph structure, called a Fault Tree (FT) is built using AND gates (represented iconically by a rounded connector) and OR gates (represented iconically by a pointed connector) to indicate how the effect of the incorrect performance of a step can propagate and cause consequent incorrect performance of other steps. The analyst must specify a particular hazard that is of concern, where a hazard is defined to be a condition that creates the possibility of loss of life or substantial financial loss. Thus, for example, the delivery to a critical step of an incorrect artifact creates a hazard if the performance of the step would then cause loss of life or substantial financial loss.

Once such a hazard has been specified, FTA can then be used to identify which combinations of incorrect step performances could lead to the occurrence of the specified hazard. Of particular interest are situations in which the incorrect performance of only one step can lead to the creation of a hazard. Such a step is referred to as a *single point of failure*. A single point of failure in a method or process creates a particularly worrisome vulnerability. Thus identification of the existence of single points of failure should be taken as an indication of the need to modify the method or process to remove such single points of failure, thereby effecting important improvement to the method or process.

Most previous work has focused on the use of FTA to identify and remove defects that lead to hazards, but have focused correspondingly less attention on how FTs are constructed. The need to be sure that an FT is complete and correct is acute as any defect in the FT can cause FTA to be incorrect, perhaps leading to overlooking an important way in which a hazard can arise. To address this problem, in earlier work [Chen 2006] it has been shown that FTs can be generated automatically from a Little-JIL process definition once the definition has been developed and a hazard has been specified. The FT and hazard specification can then be used to identify such vulnerabilities as single points of failure.

We now show that the technology described in [Chen 2006] can be applied to the identification of single points of failure in methods such as the Scrum. We use this example to indicate how a single point of failure can be removed. In this particular example we do so by integrating Scrum with another method, thus also showing how the manipulation of rigorously defined processes can be done with greater assurance of the nature of the results of such manipulation.

Identification of a Scrum method hazard.

A key element of Scrum is the notion that at the end of each 30 day Sprint, the *ScrumMaster* presents the *product* that the team has built. However, while it would surely not be something that any Scrum practitioner would endorse, "write code for the first 29 days and then only on the 30th day make a first attempt to integrate everything" is not at all inconsistent with the definition of the Scrum method. We hasten to note that this should not be perceived as a weakness of Scrum, but only an observation about the intended scope of the issues that Scrum is intended to address. We observe that Scrum explicitly states that it is intended as a management approach, and that it does not specify any engineering processes. It is expected that performance of Scrum will be integrated with appropriate engineering processes intended to support the instilling of desired properties (e.g. product quality, process speed) into the product being created. We now demonstrate how the above mentioned vulnerability can be identified by building and analyzing an FT generated from the Scrum definition. We then demonstrate how integrating the Scrum method so defined with the definition of a Continuous Integration process can lead to an integrated method which removes the single point of failure, leading to a process that does not have the original vulnerability.

A complete treatment of the way in which an FT is generated from a Little-JIL definition is beyond the scope of this document, but the interested reader is referred to [Chen 2006] for more details. Suffice it to assert, however, that automated tools applied to the Scrum definition leads to the generation of the FT shown in Figure 14.

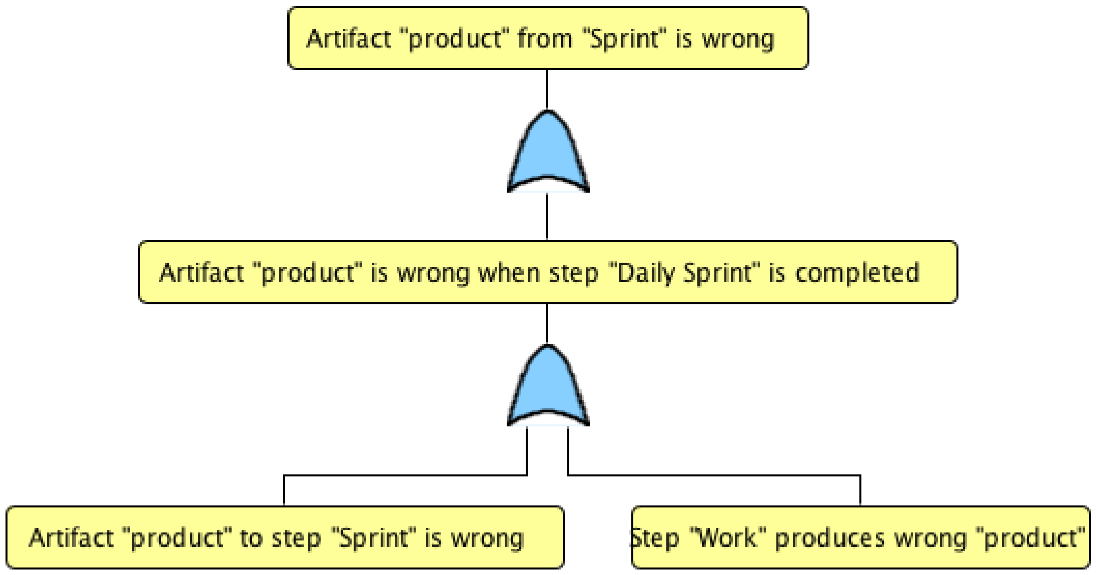

Artifact "product" from "Sprint" is wrong
Artifact "product" is wrong when step "Daily Sprint" is completed
Artifact "product" to step "Sprint" is wrong
Step "Work" produces wrong "product"

Figure 14. The Fault Tree generated by the Scrum definition, along with the hazard, "Artifact *product* from Sprint is wrong".

Note that the gates shown in Figure 14 are OR gates (represented by pointed connector icons). Thus this FT specifies that the incorrect performance of the **Work** step is sufficient to create the hazard that the artifact produced by **Sprint** will be incorrect. Thus any instance of the (multiply instantiated) Work step in the Scrum method defined in Figure 8 is a single point of failure. Presumably this is a vulnerability that should be removed.

We propose that this vulnerability be removed by integrating the Continuous Integration method into the previous description of the Scrum process.

Modeling Continuous Integration

To integrate the Continuous Integration method with the Scrum method (Figure 15), we replace the step **Work** in the original Scrum definition, with a sub-process **Checked Work**. In **Checked Work**, the original **Work** step is followed by the **Integrate** step, whose purpose is specifically to integrate the work just completed with prior work products. The successful integration of this new work is verified by the performance of a post-requisite to the **Integrate** step that verifies the correctness of the integrated artifact. If the verification does not succeed, then the **Rework** step is performed, to make whatever modifications are necessary to ensure that the required modification are carried out. Details of the Continuous Integration method are not provided here, as the purpose of this section is to indicate how process rigor can support greater assurance about the success of method integration. Details of the Continuous Integration method definition would look very analogous to the details of the Scrum method definition.

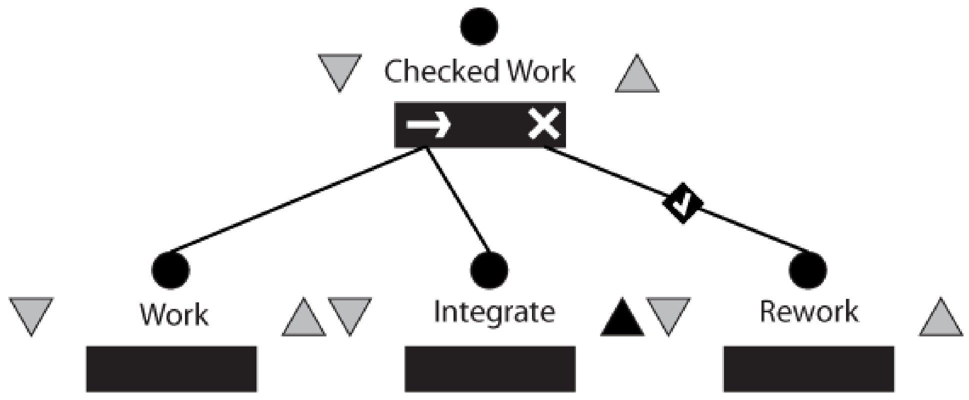


Figure 15. Integration of Continuous Integration into the Scrum process definition

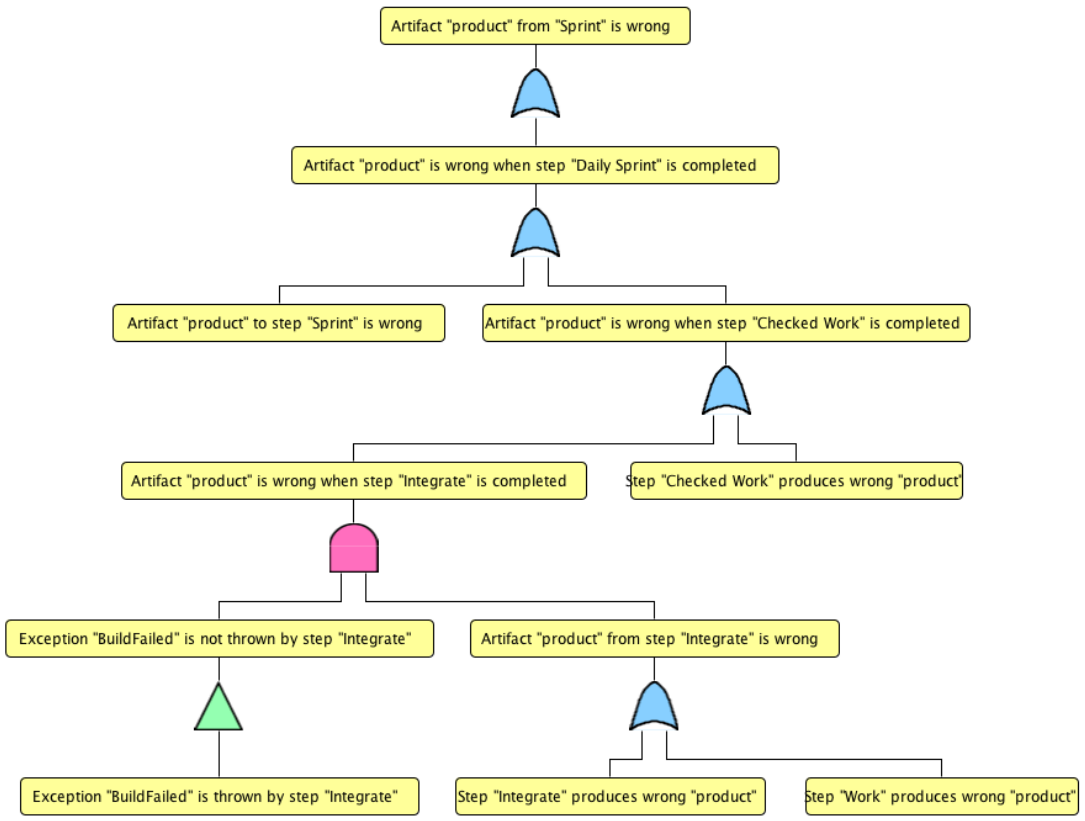

Artifact "product" from "Sprint" is wrong
Artifact "product" is wrong when step "Daily Sprint" is completed
Artifact "product" to step "Sprint" is wrong
Artifact "product" is wrong when step "Checked Work" is completed
Artifact "product" is wrong when step "Integrate" is completed
Step "Checked Work" produces wrong "product"
Exception "BuildFailed" is not thrown by step "Integrate"
Artifact "product" from step "Integrate" is wrong
Exception "BuildFailed" is thrown by step "Integrate"
Step "Integrate" produces wrong "product"
Step "Work" produces wrong "product"

Figure 16. The Fault Tree generated from the integrated Scrum and Continuous Integration processes using the hazard, "Artifact *product* from Sprint is wrong".

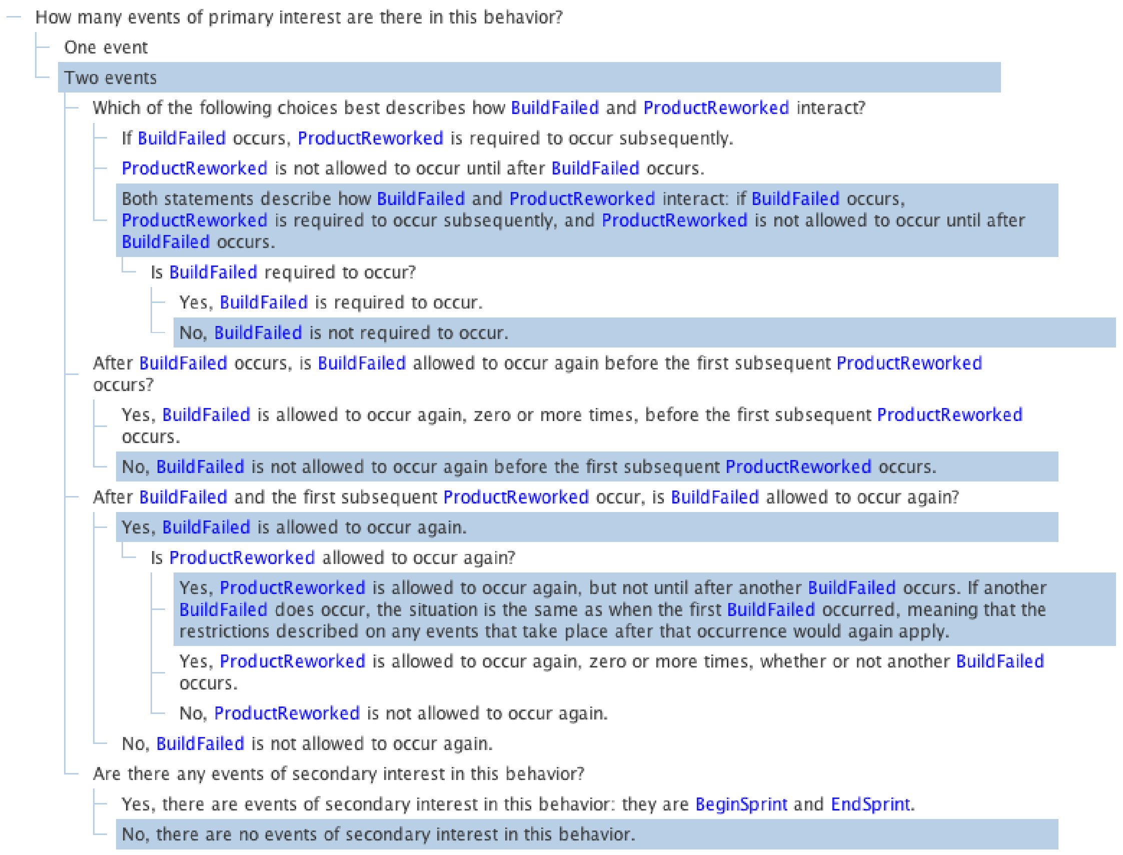
To conclude this demonstration, we now show the Fault Tree generated from the process definition as modified by the integration of the subprocess shown in Figure 16, using the same hazard used to generate the Fault Tree shown in Figure 14.

Analysis shows that there is no longer any single point of failure step in this Fault Tree. For this modified method, if "Step **Work** produces the wrong *product*", then it will still be necessary for "Exception 'BuildFailed' is thrown by step 'integrate' " to fail to be thrown (note that the icon connecting "Exception 'BuildFailed' is thrown by step 'integrate' " to its parent is a complementation gate). Thus two steps must fail to be performed correctly in order for the hazard to occur. In this way an identified vulnerability has been removed by the integration in an appropriate way of a new method into a prior method.

Using Finite State Verification to demonstrate the absence of defects.

Finite-state verification techniques are widely used to demonstrate the absence of specified event sequence defects from computer hardware and from software code or designs. In [Chen 2008] it is shown that Finite state verification can also be used to demonstrate the presence or absence of such defects from definitions of methods and processes that are sufficiently rigorously defined. In this section we now show how this analysis technology can be used to check if user defined properties hold in the Little-JIL Scrum model presented above.

The example event sequence property that we will study is that an integration failure is always followed by rework to correct the problem. For this work we have used the FLAVERS finite-state verification tool [Dwyer 2004], which uses finite-state machines to specify properties that are most often created using a tool PROPEL that allows the use of guided questions and disciplined natural language to aid the user in the creation of the property specification [Cobleigh 2006]. To specify our property, we have answered the guided questions from PROPEL as shown in Figure 17:



- How many events of primary interest are there in this behavior?
  - One event
  - Two events
    - Which of the following choices best describes how BuildFailed and ProductReworked interact?
      - If BuildFailed occurs, ProductReworked is required to occur subsequently.
      - ProductReworked is not allowed to occur until after BuildFailed occurs.
      - Both statements describe how BuildFailed and ProductReworked interact: if BuildFailed occurs, ProductReworked is required to occur subsequently, and ProductReworked is not allowed to occur until after BuildFailed occurs.
        - Is BuildFailed required to occur?
          - Yes, BuildFailed is required to occur.
          - No, BuildFailed is not required to occur.
    - After BuildFailed occurs, is BuildFailed allowed to occur again before the first subsequent ProductReworked occurs?
      - Yes, BuildFailed is allowed to occur again, zero or more times, before the first subsequent ProductReworked occurs.
      - No, BuildFailed is not allowed to occur again before the first subsequent ProductReworked occurs.
    - After BuildFailed and the first subsequent ProductReworked occur, is BuildFailed allowed to occur again?
      - Yes, BuildFailed is allowed to occur again.
        - Is ProductReworked allowed to occur again?
          - Yes, ProductReworked is allowed to occur again, but not until after another BuildFailed occurs. If another BuildFailed does occur, the situation is the same as when the first BuildFailed occurred, meaning that the restrictions described on any events that take place after that occurrence would again apply.
          - Yes, ProductReworked is allowed to occur again, zero or more times, whether or not another BuildFailed occurs.
          - No, ProductReworked is not allowed to occur again.
      - No, BuildFailed is not allowed to occur again.
    - Are there any events of secondary interest in this behavior?
      - Yes, there are events of secondary interest in this behavior: they are BeginSprint and EndSprint.
      - No, there are no events of secondary interest in this behavior.

Figure 17. Progression of questions and answers used by the PROPEL tool to evolve a finite state automaton precisely defining the property, "an integration failure is always followed by rework".

To generate this property:

1. The events of primary interest in this behavior are BuildFailed and ProductReworked
2. If BuildFailed occurs, ProductReworked is required to occur subsequently.
3. Before the first BuildFailed occurs:
   - ProductReworked is not allowed to occur.
4. BuildFailed is not required to occur.
5. After BuildFailed occurs, but before the first subsequent ProductReworked occurs:
   - BuildFailed is not allowed to occur again.
6. After BuildFailed and the first subsequent ProductReworked occur:
   - ProductReworked is not allowed to occur again until after another BuildFailed occurs;
   - BuildFailed is allowed to occur again and, if it does, then the situation is the same as when the first BuildFailed occurred, meaning that the restrictions described in parts 2, 5, and 6 would again apply.

The finite state automaton generated by PROPEL for use with FLAVERS is then:

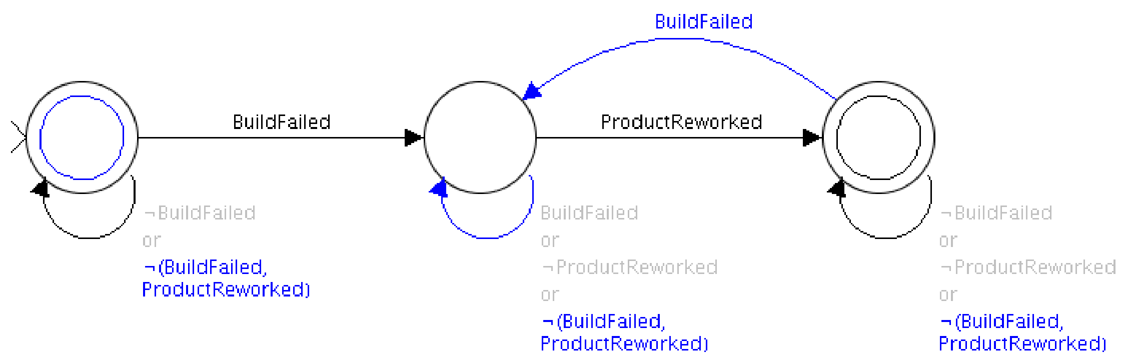


Figure 18. The Finite State Automaton generated by the PROPEL session shown in Figure 17.

To specify that reworking the system involves re-executing the step **Integrate** to fix the integration errors, we elaborate the step **Rework** shown in Figure 16 as shown in Figure 19.

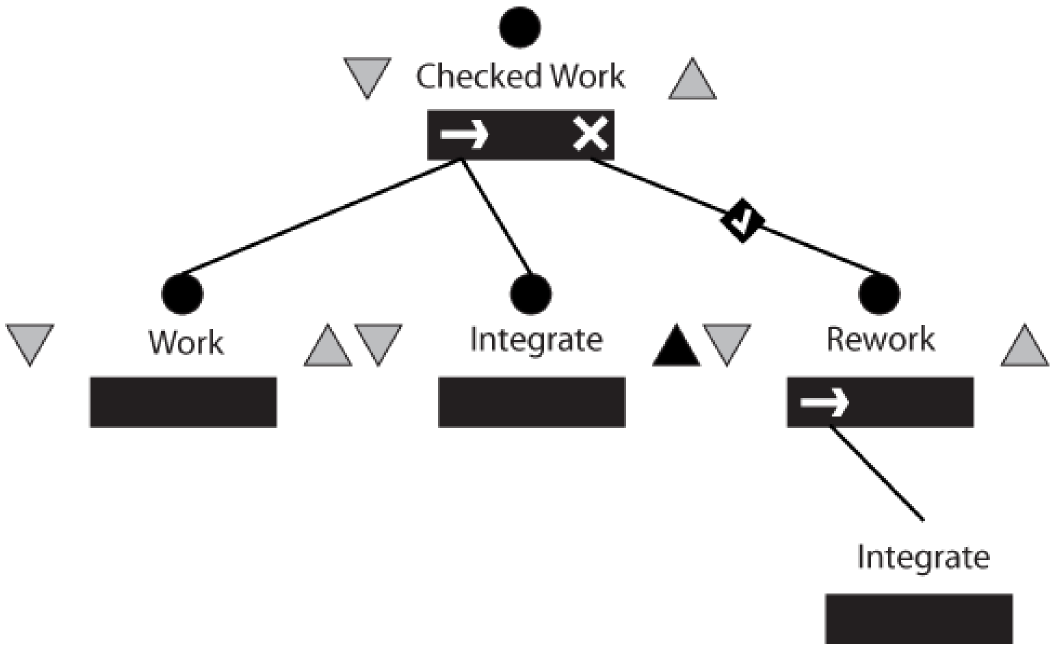


Figure 19. Elaboration of the Checked Work process shown in Figure 18 to show details about the nature of the Rework step.

FLAVERS analysis of this process definition reveals that the desired property does not hold: in the event that rework still does not result in a properly integrated system, the postcondition on the Integrate step will be evaluated again, but there is no exception handler specified on the Rework step to catch this exception. Thus, the rework is not repeated a second time even if the first rework attempt proves to be unsuccessful. This is a common subtle error in processes that involving rework. The error is readily corrected by adding the missing exception handler to the Rework step.

The above example demonstrates the use of finite-state verification to identify problems in method and process definitions, and to indicate how these defects might be removed. For example, checking the property that every development iteration begins with a **Sprint Planning Meeting** and ends with a **Sprint Retrospective** reveals that although **Scrum** specifies several time-boxed tasks, it does not specify what to do if the tasks are not completed within the allotted time. While this oversight is minor and surely would be handled in an intelligent manner by the Scrum participants it nevertheless shows how exceptional situations are commonly left out of informal process specifications and may be easily identified through analysis such as is described here.

REFERENCES

[Cass 2000] Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Stanley M. Sutton, Jr., Alexander Wise. “Little-JIL/Juliette: A Process Definition Language and Interpreter,” 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, pp. 754-757, June 2000. *(UM-CS-2000-066)*

[Chen 2006] Bin Chen, George S. Avrunin, Lori A. Clarke, Leon J. Osterweil. “Automatic Fault Tree Derivation from Little-JIL Process Definitions,” 2006 Software Process Workshop (SPW 2006) and 2006 Process Simulation Workshop (PROSIM 2006), Shanghai, China, Springer-Verlag LNCS, Vol. 3966, pp. 150-158, May, 2006. *(UM-CS-2006-01)*

[Chen 2008] Bin Chen, George S. Avrunin, Elizabeth A. Henneman, Lori A. Clarke, Leon J. Osterweil, Philip L. Henneman. “Analyzing Medical Processes,” ACM SIGSOFT/IEEE 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 2008, pp. 623-632. *(UM-CS-2007-51)*

[Cobleigh 2006] Rachel L. Cobleigh, George S. Avrunin, Lori A. Clarke. “User Guidance for Creating Precise and Accessible Property Specifications,” ACM SIGSOFT 14th International Symposium on Foundations of Software Engineering (FSE14), Portland, OR, pp. 208-218, November 2006. *(UM-CS-2006-27)*

[Dwyer 2004] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, Gleb Naumovich. “Flow Analysis for Verifying Properties of Concurrent Software Systems,” ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 13, No. 4, pp. 359-430, October 2004. *(UM-CS-2004-006)*

[Mccall 1998] Eric K. McCall, Lori A. Clarke, Leon J. Osterweil. “An Adaptable Generation Approach to Agenda Management,” 20th International Conference on Software Engineering (ICSE 1998), Kyoto, Japan, pp. 282-291, April 1998. *(UM-CS-1997-045)*

[Wise 2006] Alexander Wise. “Little-JIL 1.5 Language,” Report Department of Computer Science, University of Massachusetts, Amherst, MA 01003, October 2006. *(UM-CS-2006-51)*

APPENDIX D – IMPLEMENTATION GUIDANCE

PACKAGED METHOD: SCRUM

Scrum can be considered more of a management approach than a technical one: It is a lightweight project management methodology that only addresses project management and planning; specific technical processes are left up to the team. Scrum was originally designed for smaller teams composed of a ScrumMaster (project manager), the product owner (stakeholder representative), and the development team. Requirements are determined by input from all stakeholders and broken down into a set of (preferably) independent features that can be completed in 2-4 weeks. The features, along with their estimated time to completion, are kept in a product backlog. Analysis, development and testing take place in 2-4 week iterations called sprints. At the beginning of each sprint, the product owner, project manager, and team determine which features can be completed during the sprint, considering priority and the estimated time required to complete a feature.

During the sprint, the project manager ensures the team is focusing on their assigned tasks and enforces accurate reporting of the amount of time spent on each feature. The specific practices and techniques used during analysis, implementation, and testing during the sprint are at the discretion of the development team. Each day, the team gathers for a daily stand-up meeting where each member describes what he/she completed the previous day, what he/she plans to do today, and anything that is blocking the team member from completing his/her goal.

At the end of each sprint, the team updates the product backlog to reflect the time spent on each feature and the time remaining on each feature. The team and stakeholders meet to discuss problems and difficulties and to plan the next sprint accordingly.

Why we recommend it

Scrum is a good fit to the sponsor's environment, given that teams need to get working systems developed on tight schedules and with constantly evolving stakeholder needs. The aim of Scrum is to provide visibility into system development progress and to use constant feedback loops to optimize the system that is built given the available resources and constraints.

Our assessment was borne out by the fact that Scrum was one of the most-often mentioned methods on our survey of the state-of-the-practice of system engineering agility. Survey respondents mentioned Scrum as helping to address three of the four primary challenges we identified in the sponsor environment (namely, changing requirements priorities and/or emerging requirements; obtaining useful stakeholder input and dealing with conflicting stakeholder requirements; and dealing with conflicts between developing new capabilities and supporting a currently deployed system).

Experiences found during our interviews (discussed in Section 4.1) provide further examples of the way in which Scrum helps teams deal with emerging requirements and stakeholder feedback.

Contexts where it is suitable

Our work indicates that Scrum is likely to be an optimal match with projects that:

- Exist in unpredictable development environments. More traditional development methodologies tend to deal with unpredictability at the start of an iteration; Scrum is explicitly designed to manage sources of variation and deal with changes at any time that they occur.
- Are developing new products rather than extending existing systems [ADM2009].
- Are relatively small. An important open question is how well the Scrum processes (the daily stand-up meetings in particular) can scale up to larger teams. An often-cited recommendation is to conduct "scrums of scrums," in which for example the daily stand up meetings would be held both within and across small teams, each of which is employing the scrum practices. Our interviews found an example of such practices on a 200 person, 20 sub-team project, but the results indicated that coordination and collaboration at this scale were problematic. Other authors have addressed the need for agile practices in general and scrum in particular to scale up. Although examples of success stories are hard to come by, some general advice for tailoring scrum for large teams can be found, including the idea of starting small and increasing the number of teams being coordinated only when the current practices are proving effective [Hutson 2009]. Given our current understanding, a good rule of thumb is that scrum be easily employed by teams of up to 10 people. Beyond that number, the complications of additional communication overhead somehow need to be managed.
- Are collaborative. Teams of any size that are resistant to communication and providing visibility into day-to-day activities will also be challenged to get the most out of the stand-up meetings. Some teams have found that "...it can require quite a significant social change from the solo-oriented software development into a cross-functional team able to commit and be accountable as a team" [Marchenko 2008].

In addition, some teams have found it helpful to have personnel available to work with customers and other stakeholders in between sprints, in order to better formulate their requirements for the system. This policy helped the development teams better estimate the effort estimates and plan the sprints [Mann 2005].

Another enabler is *training*. A report on adoption of Scrum at Yahoo! India focused on the importance of training for the team and manager, with additional intensive training needed for the ScrumMaster [Sharma 2007].

Within these parameters, scrum has been effectively applied in a variety of domains. Our interviews found examples within defense, health care IT, and gaming systems, to give just a few examples.

Known impacts on cost, quality, and schedule

The available evidence for Scrum indicates a likely increase in *quality*, for example:

- In a report from Yahoo! India, 80% of teams indicated improved quality after adopting Scrum. [Sharma 2007]
- In a 2-year industrial case study, although there is no direct impact mentioned, the team anecdotally experienced positive developer and customer satisfaction: “I believe there has been far greater consistency, transparency and coordination since the implementation of Scrum” [Mann 2005]

The available evidence for Scrum indicates a likely reduction in *cost*, for example:

- In a report from Yahoo! India, 95% of teams indicated increased productivity and efficiency after adopting Scrum. The magnitude of the increase was 20%. [Sharma 2007]

The available evidence for Scrum indicates a likely reduction in *schedule*, for example:

- In a 2-year industrial case study, the team found a threefold reduction in the amount of overtime necessary for meeting development goals, after Scrum was instituted. [Mann 2005]

How to use it / monitor it

Like many other practices, teams often say they are doing Scrum when they have adopted only some of the processes and tools associated with the method. Anecdotally, many of us have encountered teams who claim to be doing Scrum but really have only adopted the process of daily stand-up meetings. Full adoption of Scrum entails additional processes to be in place, including planning meetings for the release as well as the current sprint, as well as reviews and retrospectives. Hard evidence that a team is really doing Scrum would be maintained lists of product and sprint backlogs, as well as burn-down charts showing how those lists are being worked over time.

Definitions of the inter-related processes, artifacts, and resources involved in Scrum can be found in Appendix C.

Useful resources

Teams applying Scrum may find the following resources useful:

- Active, Intelligent Management:
  http://www.controlchaos.com/about/management.php

  This site contains a good overview of terms, basic processes, and resources that may be helpful for teams getting started with Scrum. It is maintained by a Scrum consulting company, so it does contain some advertising.

- RUP in the Dialogue with Scrum:
  http://www.controlchaos.com/module/RationalEdge0205.pdf

  This article, from IBM's online magazine *The Rational Edge*, provides a basic introduction to Scrum along with ideas about how to incorporate Scrum into environments already using the Rational Unified Process.

- Scrum Guide:
  http://www.scrum.org/storage/scrumguides/Scrum%20Guide.pdf#view=fit

  This URL links to the latest version of the Scrum guide, developed by many of the people who have been instrumental in developing and popularizing the approach. The latest version at the time of writing is dated November 2009.

- Scrum Development Process (Ken Schwaber):
  http://jeffsutherland.com/oopsla/schwapub.pdf

  A more detailed overview of the Scrum process, including a comparison to spiral and incremental models, written by Ken Schwaber, who helped formulate the initial version of the Scrum process.

Sources

[ADM 2009] Advanced Development Methods, Inc., "How It Works," http://www.controlchaos.com/about/how.php, retrieved on 2009-12-13.

[Hutson 2009] Hutson, S., and Componation, P. 2009. "Agile Software Development and the Application of Its Principles to Large, Complex System Development," Unpublished white paper.

[Marchenko 2008] Marchenko, A. and Abrahamsson, P. 2008. "Scrum in a Multiproject Environment: An Ethnographically-Inspired Case Study on the Adoption Challenges," In Proceedings of Agile 2008 (August 04 - 08, 2008). IEEE Computer Society, Washington, DC, 15-26. DOI= http://dx.doi.org/10.1109/Agile.2008.77

[Mann 2005] Mann, C. and Maurer, F. 2005. "A Case Study on the Impact of Scrum on Overtime and Customer Satisfaction," In Proceedings of the Agile Development Conference (July 24 - 29, 2005). IEEE Computer Society, Washington, DC, 70-79. DOI= http://dx.doi.org/10.1109/ADC.2005.1

[Sharma 2007] Sharma, S. and Mehta, C. 2007, “Growing importance of Agile & Scrum: The Yahoo! Experience,” Bangalore SPIN, retrieved on 2009-12-13.
http://www.bspin.org/archeives11/BSPIN_BCIC_Conf_Registration.doc/view

PACKAGED METHOD: RAPID PROTOTYPING

Rapid prototyping (often called Rapid Application Development when concerning software) is the development of an executable system that implements or mimics a subset of the desired functionality of a system. The working prototype is used to uncover errors, ambiguities or omissions in the system requirements, identify usability and other non-functional concerns, improve design, and improve maintainability. By rapidly producing prototypes, developers can solicit feedback quickly and early in the development process from key stakeholders. Rapid prototyping creates *minimal* systems that demonstrate or explore functional requirements, but without the architectural robustness or non-functional properties necessary for integration into the final deliverable. Rapid prototyping can also be used to create minimal systems that exhibit desired behavior for testing against the deployment system.

Why we recommend it

Rapid prototyping is a good fit for projects that are pushing the envelope in terms of the functionality that needs to be delivered. It allows new approaches and new technologies to be tested for feasibility, with quick corrective action taken based on evaluation of intermediate versions of the system. It was created to allow faster development of system requirements, and better adapt to changing requirements.

It is important to note that there are rigorous process elements – especially those related to effectively capturing feedback from key stakeholders - that must be included for prototyping projects to be successful.

Rapid prototyping was the most-often mentioned recommendation on our industry survey, where respondents felt that it addressed three of the four challenge areas we identified (namely, changing requirements priorities and/or emerging requirements; obtaining useful stakeholder input and dealing with conflicting stakeholder requirements; and integrating systems when interoperable components are being built by different teams). The interviews reported in Section 4.1 found that rapid prototyping was commonly used by our respondents, and provide examples of how all of these challenges were being addressed by the use of rapid prototyping on projects.

Contexts where it is suitable:

Traditionally, the rule of thumb has been that the projects that can make the best use of rapid prototyping are those that require a significant degree of user interaction. User interface issues especially benefit from working through with a prototype. On the other hand, systems that are more transaction-oriented, or which are expected to have significant issues related to chronology or synchronization of functionality, are expected to benefit less [Pfleeger 2009].

Although this is commonly accepted advice, we have found instances in which rapid prototyping has been useful for systems outside of this range; for example, database-intensive systems [Lichter 1993].

Known impacts on cost, quality, and schedule:

There has been a significant amount of experience published in the literature concerning prototyping – most of it in rather early work when the concept of prototyping was just being adapted for software-engineering and software-intensive systems. As a result, there are several hard quantitative figures that can be found (although results on contemporary projects may not be exactly similar). For example:

- One study compared different versions of the same small-scale system, some built via prototyping and some based on a static specification. Results showed that systems built with each approach provided roughly equivalent functionality, but that prototyping required less effort (45% less on average than the systems built via specification) [Boehm 1984].

- A case study examined the development of a 20KLOC system in a university environment which was built around evolutionary prototyping. Regarding cost, the authors note that approximately 13KLOC were discarded as the prototype evolved into the final system (i.e. 33KLOC were developed and paid for). However, they report that the final system was of good quality (12 delivered errors in total were detected after delivery to the customer, over a period of 4 months), and that the system was easily maintainable [Hekmatpour 1987].

- A survey examined five instances of prototyping in multiple industry and manufacturing companies. Project budgets ranged from 2 to 240 person-years of effort. Regarding cost, the authors note that in a prototyping context, effort is often under-estimated for activities such as end-user evaluation of the prototype and changing / reimplementing prototypes based on that evaluation. Also, effective interaction with the end-users can be difficult; in some cases, encouraging the users to list "all the ideas and wishes that come into their heads" can result in many non-necessary functionalities making it into the requirements, driving up implementation costs [Lichter 1993].

How to use it / monitor it:

Too often, teams are tempted to dive into software development without a full understanding of what exactly they are trying to build, and call the resulting process "prototyping." Truly applying rapid prototyping means that a team should have mechanisms in place for:

- Modeling business requirements: Before embarking on development of a prototype, the team should have completed some type of analysis of the customer's business requirements. This analysis is necessary so that the development teams knows exactly what functionality is supposed to be demonstrated and evaluated in the prototype, as well as how it fits into the larger context. Such an analysis is also necessary in order to have confidence that the full range of appropriate customers / stakeholders have been taken into account.

- Obtaining useful feedback on prototypes: The effort for constructing a prototype will not have been well-spent unless there is an effective way for that prototype to generate useful feedback, which can be used to develop a comprehensive and correct set of requirements. Without an effective mechanism, the team runs the dual risks of the users not being able to visualize sufficiently what the final system should look like, or of the users providing a too-comprehensive wish list that is infeasible to satisfy [Lichter 1993].

More details on the inter-related processes, artifacts, and resources involved in rapid prototyping can be found in Appendix C.

Useful resources:

Teams applying rapid prototyping may find the following resources useful:

- Wikipedia has a useful page which provides a short overview of rapid prototyping and links to a number of tools: http://en.wikipedia.org/wiki/Software_prototyping#Best_projects_to_use_prototyping
- UsabilityNet.org introduces a brief method description, which provides an effective way of capturing user feedback on a prototype system: http://www.usabilitynet.org/tools/rapid.htm
- Microsoft's Visual Basic is one of the most commonly used tools for developing prototypes of software systems: http://msdn.microsoft.com/en-us/vbasic/default.aspx

Sources

[Pfleeger 2010] Pfleeger, S. L., and Atlee, J. (2010) Software Engineering: Theory and Practice. Upper Saddle River, NJ: Prentice Hall.

[Boehm 1984] Boehm, B. W., Gray, T. E., and Seewaldt, T. 1984. "Prototyping vs. specifying: A multi-project experiment," In Proceedings of the 7th international

Conference on Software Engineering (Orlando, Florida, United States, March 26 - 29, 1984). International Conference on Software Engineering. IEEE Press, Piscataway, NJ, 473-484.

[Hekmatpour 1987] Hekmatpour, S. 1987. “Experience with evolutionary prototyping in a large software project,” SIGSOFT Software. Eng. Notes 12, 1 (Jan. 1987), 38-41. DOI= http://doi.acm.org/10.1145/24574.24577

[Lichter 1993] Lichter, H., Schneider-Hufschmidt, M., and Züllighoven, H. 1993. “Prototyping in industrial software projects—bridging the gap between theory and practice,” In Proceedings of the 15th international Conference on Software Engineering (Baltimore, Maryland, United States, May 17 - 21, 1993). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 221-229.

PACKAGED METHOD: CONTINUOUS INTEGRATION

"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly." [Fowler 2009]

When following the process of continuous integration, developers will integrate their code with a local copy of the mainline several times daily. The local copy of the mainline includes a regression test suite including unit, integration and system tests (preferably automated) that detect integration errors locally before committing to the mainline. Developers should always obtain the latest update from the source repository and rebuild as a final check before committing to the mainline.

Once developers eliminate any integration errors, they commit their new code (and associated test cases) to the source repository on the mainline. When a commit is received, all other developers are notified that a change has been made to the mainline so that they can update their local working copies. Upon commit to the mainline, the entire system is built and deployed to an integration machine that is a clone of the production environment. The build tests are then run on the integration machine. This automated build ensures that the new code is tested against the latest codebase available, and that the build is tested in a clone of the production environment to identify environmental errors that may not be present on the development machine. If any errors are detected in the mainline, the developer fixes them and recommits.

Why we recommend it

Continuous integration is a powerful tool for detecting integration errors during development, and also for enforcing interface standards. Continuous integration also provides continuous feedback into the development process, avoiding "big bang" integrations prior to delivery that are often costly and error-prone.

Continuous integration was one of the most frequently recommended MPTs on the industry survey, and helps to address both integration/interoperability challenges and sustainment challenges. The automated testing and deployment infrastructure necessary for continuous integration reduces the development costs associated with maintenance and evolution of existing products. Implementing continuous integration in a development environment does incur initial overhead costs, however, teams will realize a long term return on their investment, particularly in projects that are maintained and evolved over an extended period.

Contexts where it is suitable

Our research indicates that continuous integration is likely to be an optimal match with projects that:

- Deliver frequently. Continuous integration eliminates "big bang" integrations prior to delivery, reducing the likelihood that final integrations will become costly and impact the delivery target. Also, continuous integration increases the likelihood that bugs will be found earlier in development, which is cheaper than fixing the bugs just prior to or during delivery. [Fowler 2009]

- Can incorporate automated testing. A necessary condition for effective continuous integration is rapid feedback. A long integration cycle that requires the developer to execute tests and deploy build environments manually increases the size of the feedback loop. Conversely, an automated test and build suite allows the developer to perform other productive functions and lessens integration effort. Furthermore, a long, effort-intensive integration cycle will discourage the developers for integrating frequently, preventing other developers from detecting conflicts in their own changes. [Fowler 2009] Our interviewees on this topic reported how difficult it was to do continuous integration manually, when they were just starting out, and felt strongly that automation was a key enabler.

- Since the cost-of-fix increases exponentially as development continues, the main benefit of continuous integration is to detect errors early and often. Continuous integration can detect interface/interoperability conflicts and errors closer to when they are introduced and before significant amounts of code is structured around the original error. For example, the cost-of-fix can increase dramatically over time in a tightly coupled system with multiple components, or in components which broadcast messages to other hardware or software systems.

Continuous integration can incur significant implementation costs driven by developing an automated testing and build framework. For new (greenfield) software projects, the return on investment for continuous integration will generally be realized sooner than for existing projects where engineering automated tests for existing code may not be viewed as cost-effective.

Our interviewees felt that projects that are not good candidates for the use of continuous integration include those for which a significant amount of hardware integration is required.

Known impacts on cost, quality, and schedule

There is little available research evidence on the efficacy of continuous integration; however, it has become a *de facto* best practice in the software industry and is used in companies such as Microsoft, Google and IBM. Investigating the implementation costs and process improvements realized by continuous integration in a systems engineering environment is an interesting and topical area for research collaboration.

Rumpe and Schröder surveyed 45 Extreme Programming teams from IT, consulting, banking and biotechnology domains – continuous integration is one of the 12 original practices of Extreme Programming. Over 90% of respondents stated that continuous integration was a "helpful" process, while on 2% found continuous integration to be difficult to implement in their teams.

How to use it / monitor it

Continuous integration requires that several key processes be in place. First, the developers must maintain a single source repository to which they all have access to and to which they all commit. Second, to encourage developers to integrate frequently and to obtain rapid feedback, the build time must be kept reasonable. This requirement necessitates an extensive automated test and build suite. Several continuous integration tools (for nearly ever language) exist to support automated test and build. Finally, the benefits of continuous integration are only realized if the developers integrate *frequently* by committing to the mainline at least once per day and by locally testing their changes multiple times per day.

The artifacts produced by continuous integration make it relatively easy to assess, particularly when used in conjunction with continuous integration tools such as CruiseControl. Automated test suites should grow in size as the developers commit to the mainline, and version control logs can identify the frequency with which individual developers are committing and checking out updated version of the mainline. Again, the successful implementation of continuous integration is contingent upon its technical infrastructure and the commitment of the development team to integrating frequently.

Definitions of the inter-related processes, artifacts, and resources involved in continuous integration can be found in Appendix C.

Sources

[Fowler 2009] Martin Fowler, "Continuous Integration," http://www.martinfowler.com/articles/continuousIntegration.html, retrieved on 2009-06-18.

[Duvall 2007] Paul Duvall, Steve Matyas, Andrew Glover, "Continuous Integration: Improving Software Quality and Reducing Risk," Addison-Wesley Professional, Upper Saddle River, NJ, 2007. (See also http://www.integratebutton.com/ -- the website for the book.)

[Lee 2009] Kevin Lee, "Realizing continuous integration," http://www.ibm.com/developerworks/rational/library/sep05/lee/, retrieved on 2009-12-09.

[Rumpe 2002] Bernhard Rumpe and Astrid Schröder, "Quantitative Survey on Extreme Programming Projects," Proc. 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering, pp. 95-100, 2002.

[Shore1 2009] James Shore, "Continuous Integration on a Dollar a Day," http://jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html, retrieved on 2009-12-09.

[Shore2 2009] James Shore, "Continuous Integration is an Attitude," http://jamesshore.com/Blog/Continuous-Integration-is-an-Attitude.html, retrieved on 2009-12-09.